Motion planning for a rotary-wing UAV in dynamic environments

by

Jacobus Nicolaas Lochner

Thesis presented in partial fulfilment of the requirements for the degree of

Master of Science in Engineering

at Stellenbosch University

Supervisor:

Dr. C.E. van Daalen

Department Electricaland Electronic Engineering

March 2020

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualifcation.

JN. Lochner

Signature: .....

March 2020

Date: .....

Abstract

In order for a fully autonomous unmanned aerial vehicle (UAV) to navigate safely in dynamic environments, a conflict free trajectory between an initial and goal state of a vehicle needs to be planned quickly and effectively. Most trajectory planning methods today only consider static environments and do not act on environmental changes that might occur. In order for a UAV to navigate through dynamic environments, a motion planning algorithm needs to be employed that is capable of dealing with these conditions. These motion planning algorithms are often applied in conjunction with a local planning method that ensures that the trajectories generated by the planning algorithm adhere to the dynamic constraint of the UAV.

The goal of this project is to find a motion planning algorithm that will provide safe flight for a rotary-wing UAV from an initial to a goal state in environments where dynamic obstacles exist. The motion planning algorithm implemented is based on the rapidly-exploring random tree (RRT) that is altered for dynamic environments and is called the real-time optimal RRT (RT-RRT*). Important changes were made to the RT-RRT* to increase the performance of the algorithm. The motion planning algorithm should adhere to the constraint of the vehicle, which is solved by employing a generic local planning method that make use of geometric-based motion primitives to construct trajectories that adhere to the constraints of the vehicle.

The complete motion planning algorithm is tested thoroughly in various simulated environments and the performance of the algorithm is analysed. The motion planning algorithm was proven to be effective in sparse environments but struggled in more cluttered environments with multiple dynamic obstacles. Obstacle estimation was then implemented to try and improve the motion planning algorithm for cluttered environments, which proved to be an effective solution for avoiding dynamic obstacles.

The trajectories generated by the motion planning algorithm was given to a realistic vehicle model to verify if the rotary-wing UAV will be able to accurately follow the generated trajectory. The vehicle model and controllers was previously designed and verified to be accurate during practical flight

tests, which means that this model is an accurate representation of a real world vehicle.

Uittreksel

Vir 'n outonome onbemande lugvoertuig om veilig te navigeer in dinamiese omgewings, word dit vereis dat 'n konflikvry trajek tussen twee voertuig toestande vinnig en effektief beplan moet word. Meeste trajek beplannings metodes van vandag, is ontwerp om beplanning te doen in statiese omgewings en kan nie reageer op veranderinge wat in die omgewing kan plaasvind nie. Dus, vir n Onbemande lugvoertuig om veilig te navigeer deur dinamiese omgewings, is dit nodig om 'n bewegingsbeplannings algoritme te ontwerp wat hierdie kondisies kan hanteers. Beplannings algoritmes word dikwels in kombinasie met plaaslike beplannings metodes geimplimenteer sodat die trajekte wat gegenereer word, voldoen aan die dinamise beperkings van die voertuig.

Die doel van hierdie projek is om 'n bewegingsbeplannings algoritme te ontwerp wat sal sorg dat 'n roterende vleuel Onbemande lugvoertuig veilig kan beweeg vanaf 'n aanvaklike tot 'n doel posisie deur 'n omgewing met dinamiese voorwerpe. Die bewegingsbeplannings algoritme wat geÃ¯mplementeer is, is gebaseer op die "Rapidly exploring Random Tree" (RRT) wat verander is om dinamiese omgewings te hanteer en word die "Real-Time RRT-star" (RT-RRT*) genoem. Belangrike veranderinge was dan gemaak op die RT-RRT* om die uitvoering daarvan te verbeter. Die finale beplannings algoritme moet wel voldoen aan die dinamiese beperkings van die voertuig, wat opgelos is deur die implimenteering van 'n generiese plaaslike beplannings metode, wat gebruik maak van geometriese bewegins-primitiewe om 'n trajek te skep wat by die beperkings van die voertuig hou.

Die finale beplannings algoritme word dan deeglik getoets in verskeie gesimuleerde omgewings sodat die uitvoer daarvan geanaliseer kan word. Dit was bewys dat die bewegingsbeplannings algoritme effektief werk vir omgewings met min voorwerpe, maar sukkel in besige omgewings waar daar verskeie dinamiese voorwerpe is. Hindernisberaming was toegepas om die algoritme te verbeter in besige omgewings en was bewys dat dit 'n effektiewe oplossing is om dinamiese voorwerpe te vermy.

Die trajekte wat geskep word deur die beplannings algoritme was aan 'n realistiese voertuig model gestuur om te varifieer dat die roterende vleuel Onbe-

mande lugvoertuig hierdie trajekte akkuraat kan uitvoer. Die voertuig model en beheerders was voorheen ontwerp en getoets tydens praktiese vlug en is dus 'n geldige voorstelling van 'n regte wÃªreld voertuig.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations and Acronyms

DCM	direct cosine matrix
GPS	Global Positioning System
IMU	inertial measurement unit
LIDAR	light detection and ranging
LPM	local planning method
LRL	left-right-left
LSR	left-straight-right
LSL	left-straight-left
NED	north-east-down
PRM	probabilistic roadmap
RLR	right-left-right
ROS	robot operating system
RRT	rapidly-exploring random tree
RRT*	optimal RRT
RSL	right-straight-left
RSR	right-straight-right
RT-RRT*	real-time optimal RRT
SLAM	simultaneous localisation and mapping
UAV	unmanned aerial vehicle

Symbols

α	angle of attack
β	angle of sideslip
$\dot{\delta}$	actuator
θ	pitch angle
ϕ	roll Angle
ψ	yaw angle
μ	mean
ρ	air density
σ	standard deviation
τ	constant time-step
A_x	acceleration in reference axis x
C	configuration space
d	wingspan
$\mathbf{G}$	controllability Gramian
g	gravitational acceleration
$\mathbf{I}$	identity matrix
m	aircraft mass
q_i	sampled node with reference name i
L, M, N	magnitude of the Roll, Pitch and Yaw moments
P, Q, R	angular velocity components
U, V, W	velocity components
X, Y, Z	directions of force components

Notation

x	scalar
$\mathbf{x}$	vector
$\mathbf{X}$	matrix
$\dot{x}$	first derivative of x with respect to time
$\mathbf{x} \sim \mathcal{N}(\mu, \sigma)$	normal distribution with mean μ and standard deviation σ

Subscripts and Superscripts

0	initial
B	body axes
I	inertial Axes
W	wind Axes
k	discrete time index

Acknowledgements

I would like to express my sincere gratitude to the following people:

- Dr. C.E. van Daalen, for his guidance, encouragement and endless supply of knowledge throughout the course of this project.
- Ryno Swart for his valued input and feedback on every aspect of my project.
- All menders of the ESL Autonomous Navigation group for the continued support and the willingness to help.
- To my friends and family for all the support and motivation.

Chapter 1

Introduction

1.1 Background

The autonomous navigation problem, as a whole, has been one of the main topics of research in the field of robotics. Applications such as autonomous driving cars and quad-rotor delivery services made headlines in the past few years and have become a popular field of development. Unmanned aerial vehicles (UAVs) are much more common in the current robotics market due to its simplicity and wide range of applications, which includes operations like surveillance, search and rescue, delivery and many more.

Most of the active UAVs today, are not seen as fully autonomous systems, in the sense that they require some degree of human intervention to safely navigate the environment they are placed in. The main drawbacks of human intervention, is the unreliability and performance of these systems, especially in environments where moving obstacles exist. This leads to the need of having an in flight path planning solution that will allow these UAVs to find collision free trajectories through the environment by making use of the multiple sensors onboard such vehicles.

Much research has been done to find motion planning solutions for various kind of vehicles and the results found, especially for environments with fixed static obstacles, was shown to be highly effective. However, motion planning solutions for environments with dynamic obstacles, is not as established.

The focus for this project is to investigate motion planning for UAVs in dynamic environments to try and find a solution that will allow safe flight through environments with dynamic obstacles. It was decided, that the motion planning solution should aim to function on-board a rotary-wing UAV, specifically a quad-rotor.

Quad-rotors, in the past few years, have become much more common especially after Amazon announced their quad-rotor-based parcel delivery system, Prime Air, in 2016, due to the simplicity and accessibility [5]. Therefore, it was decided to make use of a quad-rotor as testing vehicle.

The next section will provide an overview of the autonomous navigation problem to provide some context on the role motion planning plays in the fully autonomous navigation system.

1.2 The Autonomous Navigation Framework

Figure 1.1 shows a proposed framework of an integrated autonomous navigation system, in order to provide some context on part motion planning plays in the broader scope of autonomous navigation. This framework shows how various modular components of the navigation system interact with one another.

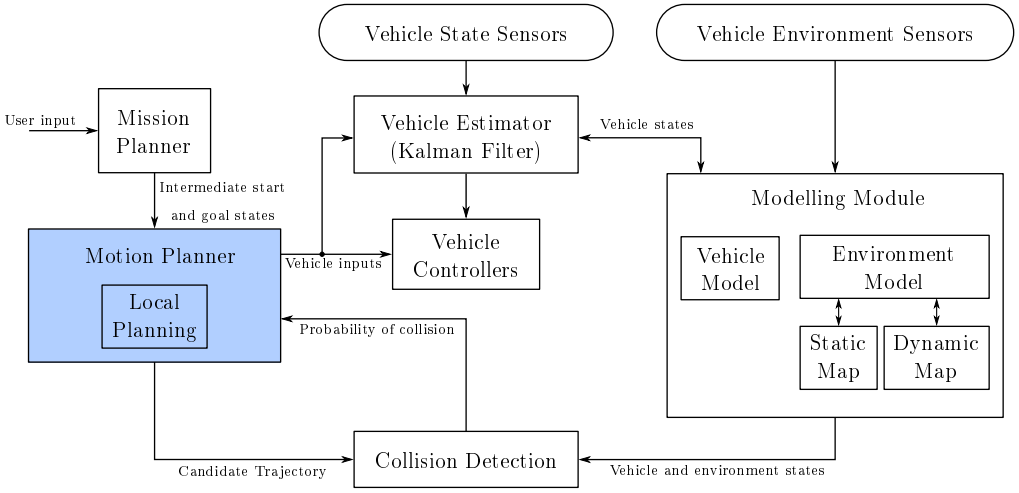


Figure 1.1: The proposed framework of an autonomous navigation system with the project scope in blue. The image was adapted from van Daalen [1].

The vehicle estimator, controllers and states sensors blocks from Figure 1.1, form part of a typical vehicle control system. The vehicle state estimator uses information gathered by the vehicle sensors to estimate the position and orientation of the vehicle. These sensors can include items like a global positioning system (GPS) sensor and inertial measurement unit (IMU). The information provided by the estimator and the data from the vehicle environment sensors can be used to estimate the vehicle states relative to the environment. An example of such a sensor is a Light Detection and Ranging (LIDAR) sensor, which uses a laser, to sense the distance of the vehicle relative to objects in the

environment. The information from the environment sensors and the vehicle state estimation can then be combined to construct a map of the environment as well as the location of the vehicle within this map. An algorithm such as simultaneous localization and mapping (SLAM), can be used for this operation. The map representation of the environment, which includes static and dynamic obstacles, will be maintained and updated by the modelling module as the vehicle moves through the environment.

The motion planner, which is developed in this project, should be able to plan a trajectory between an initial and goal vehicle state by making use of the map of the environment, obtained from the modelling module. The motion planner requests a collision analysis from the collision detection module, by providing it with a candidate trajectory. The collision detection module make use of the estimated vehicle states and the environment representation from the modelling module to calculate the probability of collision along the given trajectory. If the probability of collision exceeds a predefined threshold, the trajectory will be seen as unsafe. Once the path planner finds a safe path from the initial to the goal states of the vehicle, the inputs to execute the corresponding trajectory is then given to the vehicle controllers to execute.

The mission planning module is a higher-level planner, which takes a long-term objective, given by a user as input, and translates it to a sequence of intermediate goals. The idea behind this process is to make use of information like air traffic, prohibited air zones and weather to generate waypoints beforehand for the vehicle to reach and make use of the motion planner to find a safe path between these waypoints.

Given that context has been provided regarding motion planning in the full autonomous navigation problem, the objectives of this project can be laid out.

1.3 Research Objectives

The primary objective of this project is to develop a motion planning algorithm that is able to construct feasible collision free trajectories in environments with dynamic and static obstacles. This algorithm should be designed for a quad-rotor, but has to be developed in such a way that it can easily be adapted for a wider range of vehicles. To develop such a motion planning algorithm, it is assumed that the information about the obstacles in the environment is given and can be presented in the form of a map. For the planning algorithm to function as an on-flight planner for a quad-rotor, the following requirements should be met:

1. With more computational power given to the algorithm, the probability

of finding a trajectory between any two vehicle states, if one exists, should increase.

2. The trajectories generated by the motion planning algorithm should conform to the dynamic constraints of the vehicle.
3. The final trajectory the vehicle has to follow to reach its goal should have a low probability of conflict with both static and dynamic obstacles.
4. The planning algorithm should be able to find alternative trajectories when the current planned trajectory becomes conflicted.
5. All the trajectories generated by the planning algorithm should be cost-effective in terms of its length.

The final objective of this project, is thus to verify the execution of the planned trajectories, by using a realistic vehicle model and controllers.

1.4 Overview of Thesis

This section provides a brief overview of each of the chapters that follows.

Chapter 2 introduces the quad-rotor model used as testing vehicle for this project. The chapter then introduces the concepts used to design the model, including the coordinate systems used in developing the model of the vehicle. The vehicle kinetics and kinematics are discussed, as well as the moments and forces that acts on the vehicle. The chapter ends of by introducing the quad-rotor model and the controllers.

Chapter 3 discusses the motion planning problem. The chapter provides a brief overview of the different types of motion planning algorithms before giving a more in-depth discussion on sampling-based motion planning and why it was chosen. The chapter then identifies that the rapidly-exploring random Tree (RRT), with certain adjustments, is able to meet the requirements set out by this project. The necessary adjustments are developed and the chapter ends with an in-depth discussion of the final motion planning approach that will be implemented.

Chapter 4 identifies two ways in which to construct a local planning method (LPM). The first one is a optimal local planner, which uses the model of a vehicle to find the best possible trajectory between any two vehicle states by minimizing a given cost function. The second approach is a geometric manoeuvre based local planning method, that is able to connect two vehicle states together with geometrically accurate motion primitives. Both these methods are discussed in more detail before deciding on which approach to

implement. The chapter ends of by revisiting the control system from Chapter 2 and makes necessary improvements based on the shortcomings of the LPM.

Chapter 5 combines the motion planning algorithm from Chapter 3 and the LPM from Chapter 4. The planner is tested in multiple environments to see if this solution meets the requirements set out in Section 1.3. Some changes are then made to the planning algorithm and retested to find out if it improves the overall performance of the planning solution. Obstacle estimation and future prediction is then added to the planner to see if it will improve obstacle avoidance.

Chapter 6 then concludes with the notable results produced by this project and proposes areas to consider investigating in the future.

Chapter 2

Aircraft Dynamics and Controllers

Before a motion planning algorithm can be developed, it is important to have an accurate representation of the chosen vehicle so that the execution of the trajectories generated by the motion planning algorithm can be verified. The chosen vehicle for this project is a quad-rotor named SLADe [4] which was previously designed by Peddle and verified to be accurate during practical flight tests. By having such a realistic vehicle model, it can be assumed that if the trajectories generated by motion planning algorithm is confirmed to be executable the algorithm will most likely work in a practical environment.

The mathematical models that describe the dynamics of the chosen vehicle and the control systems for stable fight are derived in this chapter. The model and controllers derived in this chapter are based on the work done by Peddle for SLADe as well as Advance Automation 833 course notes by Engelbrecht [2].

2.1 Axis Systems

The axis systems used to model the aircraft dynamics are defined in this section.

2.1.1 Inertial Axes

The north-east-down (NED) inertial axis system is typically used for short-ranged unmanned aerial vehicles (UAVs), which is shown in Figure 2.1. The inertial axis system makes use of the assumption that the earth is flat and chooses a centre point on the earth's surface at a convenient location, for example, an aircraft's take-off location. The x-axis (X_E) of the inertial axis system points north, the y-axis (Y_E) east and z-axis (Z_E) down towards the centre of the earth.

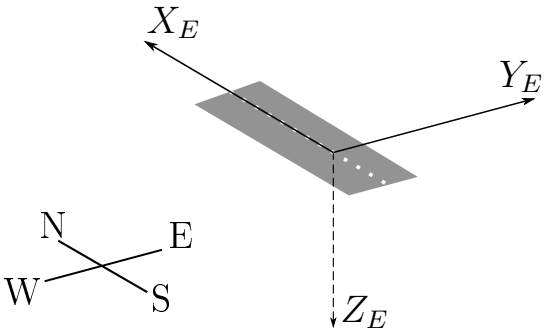


Figure 2.1: NED inertial axis system, with an airplane runway as centre point.

2.1.2 Body Axes

The body axes are fixed to the body frame of the aircraft, in this case a quad-rotor, with the origin chosen to coincide with the centre of mass of the aircraft as seen in Figure 2.2. The x-axis (X_B) is aligned with the arm of the quad-rotor pointing in the forward direction. The y-axis (Y_B) lies perpendicular to the forward arm in the direction of the right arm and the z-axis (Z_B) points downwards relative to the aircraft, which completes the right-handed axis system.

2.1.3 Wind Axes

The wind axis system, shown in Figure 2.2, is similar to the body axes, the only difference being the x-axis (X_W) that points in the direction of the velocity vector of the aircraft. The z-axis (Z_W) points in the direction of airflow through the rotors, which is normal to the x-axis, and the y-axis (Y_W) completes the right-handed axis system.

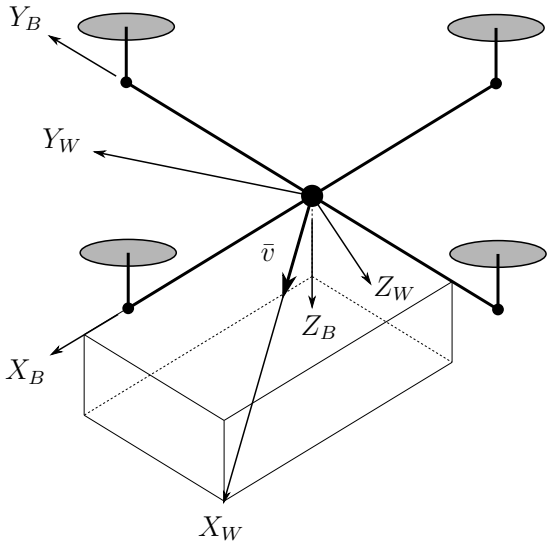


Figure 2.2: The body and wind axis systems of a quad-rotor.

2.2 Definitions

This section describes the definition of forces, moments and velocities experienced by the quad-rotor. The following symbols are used to describe aspects of the aircraft respective to the body axes, as shown in Figure 2.3.

X, Y, Z	-	The force components
L, M, N	-	The magnitude and direction of the moments
U, V, W	-	The velocity components
P, Q, R	-	The angular velocity components

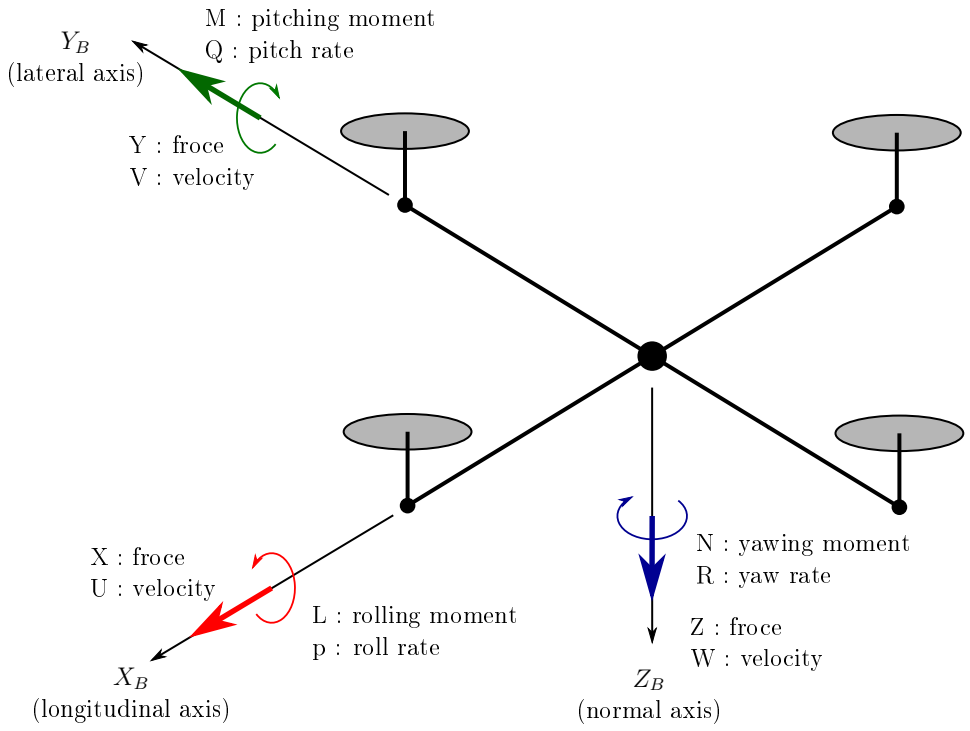


Figure 2.3: The standard aircraft notation used for modelling

The velocity vector of the aircraft is defined in spherical coordinates as shown in Figure 2.4. The velocity magnitude $\bar{V}$ is calculated from the velocity components as,

$$|\bar{V}| = \sqrt{U^2 + V^2 + W^2}, \tag{2.1}$$

the angle of attack α is given by,

$$\alpha = \tan^{-1}\left(\frac{W}{U}\right), \tag{2.2}$$

and the sideslip angle β are given by,

$$\beta = \sin^{-1}\left(\frac{V}{\bar{V}}\right). \tag{2.3}$$

The inverse relationship between Cartesian and spherical coordinates is given by:

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} \bar{V}\cos\alpha\cos\beta \\ \bar{V}\sin\beta \\ \bar{V}\sin\alpha\cos\beta \end{bmatrix}. \tag{2.4}$$

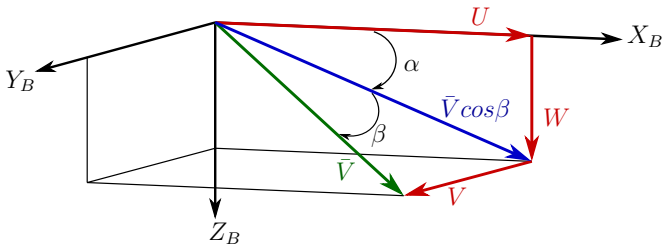


Figure 2.4: Velocity vector representation in spherical coordinates.

This concludes the definition of the forces, moments and velocities with respect to the body axis of the quad-rotor.

2.3 Kinetics

Kinetics refer to the relationship between the forces and moments acting on an object's kinematic state, such as position, velocity and acceleration. By making use of Newton's second law of motion, these relationships can be derived as follows:

$$\begin{aligned} X &= m(\dot{U} - VR + WQ) \\ Y &= m(\dot{V} + UR - WP) \\ Z &= m(\dot{W} - UQ + VP) \\ L &= \dot{P}I_{xx} + QR(I_{zz} - I_{yy}) \\ M &= \dot{Q}I_{yy} + PR(I_{xx} - I_{zz}) \\ N &= \dot{R}I_{zz} + PQ(I_{yy} - I_{xx}). \end{aligned} \tag{2.5}$$

where m is the mass of the aircraft and I_{xx}, I_{yy} and I_{zz} are the moments of inertia about the body axes on each axis respectively. The following assumptions were made to simplify the equations above:

- The aircraft is symmetrical about the XZ-plane, which implies that the I_{xy} and I_{yz} inertial components are zero.
- The I_{xz} inertial component is negligibly small.
- The aircraft is a rigid body with a constant mass.

These assumptions are considered to be very accurate for all conventional aircraft and thus valid for the purpose of this project. This end of the representation of the quad-rotor's kinetics and the following section will now looks at the quad-rotor's kinematics.

2.4 Kinematics

The kinematics of an aircraft describe the mathematical relationship between the position, attitude (angular position), velocity and angular velocity to each other. These relationships are necessary to define the transformation from the inertial axes to the body axes. The following subsection addresses the Euler 3-2-1 parameterisation for aircraft attitude, as proposed by Engelbrecht [2] .

2.4.1 Euler Attitude Parameterisation

When describing the motion of the aircraft in the inertial axis system, it is necessary to describe the orientation of the aircraft in the body axis system relative to the inertial axis system. This can be done through the means of Euler angle parameterisation. Euler angle parameterisation uses three angles to describe the attitude of the body axis system relative to the inertial axis system. For this project, the 3-2-1 Euler angle sequence is adopted, which starts with the two axis systems aligned and then sequentially performs the following rotations:

- a yaw rotation of the aligned body axis system through the heading angle ψ.
- a pitch rotation around the yawed body axis system through the pitch angle θ.
- a roll rotation around the yawed and pitched body axis system through the roll angle ϕ.

A graphical representation of the 3-2-1 Euler rotations in sequence can be seen in Figure 2.5, which converts the inertial axis system to the body axis system.

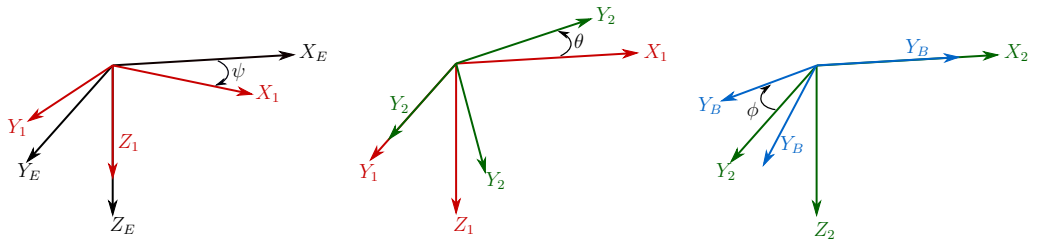


Figure 2.5: Euler angle rotations of the body axis system relative to the inertial axis system.

When the Euler angle rotations are used in sequence, it will be possible to relate a vector in one axis system to the same vector in another axis system, which is needed to construct a transformation matrix that will be able to relate the quad-rotors velocities from the body axis to the inertial axis system. This matrix known as the the direct cosine matrix (DCM) and is presented in Section 2.4.3.

The attitude of the body axes relative to the inertial axes is thus fully captured by Euler angles. Note the singularity that occurs at $\pm 90°$ pitch angle. This, however, can be ignored due to the fact that during conventional flight, the pitch angle will not reach $\pm 90°$. These parameters can now be used to represent the quad-rotors attitude and position dynamics.

2.4.2 Attitude Dynamics

By expressing the relationship between the angular rate of the aircraft's body (P, Q, R) and the rate of change of the Euler angles, the attitude dynamics of the quad-rotor can be expressed by:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{bmatrix} \begin{bmatrix} P \\ Q \\ R \end{bmatrix} \qquad |\theta| \neq \frac{\pi}{2}, \tag{2.6}$$

which describes how the roll, pitch and yaw rates change relative to the roll pitch and yaw angles.

2.4.3 Position Dynamics

The last kinematic equations to derive are the position dynamics, which is the speed at which the aircraft moves in the inertial axes in terms of the aircraft's velocity in the body axis system. The relationship between the position and velocity vectors in the inertial axes is presented as,

$$\begin{bmatrix} \dot{N} \\ \dot{E} \\ \dot{D} \end{bmatrix} = \begin{bmatrix} V_N \\ V_E \\ V_D \end{bmatrix}, \tag{2.7}$$

where, V_N, V_E and V_D are the velocities in north, east and down respectively. However, up until this point, the dynamics have been derived as a function of the body axes velocity coordinates (U, V, W). Thus, a transformation matrix should be constructed that is able to relate the velocity vector in one axis system to the same vector in another axis system. This transformation matrix, as mentioned in Section 2.4.3, is called the DCM matrix. To construct the DCM matrix, the vector in the original axis system (inertial axes) is rotated with yaw, pitch and roll angle, which yields

$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{N} \\ \dot{E} \\ \dot{D} \end{bmatrix}$$
$$\begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} C_\psi C_\theta & S_\psi C_\theta & -S_\theta \\ C_\psi S_\theta S_\phi - S_\psi C_\phi & S_\psi S_\theta S_\phi + C_\psi C_\phi & C_\theta S_\phi \\ C_\psi S_\theta C_\phi + S_\psi S_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi & C_\theta C_\phi \end{bmatrix} \begin{bmatrix} \dot{N} \\ \dot{E} \\ \dot{D} \end{bmatrix} \tag{2.8}$$

where $C_* = \cos(*)$ and $S_* = \sin(*)$. For the position dynamics of the aircraft, it is necessary to convert the velocity components in the body axis system to the inertial axis system. Thus, the inverse DCM matrix is required. In this case, due to the DCM matrix being orthogonal, the inverse matrix its simply the transpose. With the transpose of Equation 2.8 it is now possible to calculate the velocity of the quad-rotor in the inertial axis.

Equation 2.5, 2.6 and 2.8 then fully introduces the six degree of freedom equations of motion for a rigid body. The next section will introduce the forces and moments acting on the quad-rotor.

2.5 Forces and Moments

With the equations of motion of the quad-rotor described in the previous section, the forces and moments that act on the aircraft can now be derived as a function of its current state. For most aircraft's these forces and moments come from, aerodynamics, actuators and gravity and will be discussed in this section.

2.5.1 Aerodynamic Lift and Drag

The aerodynamic forces and moments acting on the aircraft are described by the drag and lift experienced by aircraft while moving through a fluid (in this

case air). The equation describing the drag experienced by the aircraft is calculated as

$$\bar{D} = \frac{1}{2}\rho v^2 A_D C_D, \tag{2.9}$$

where,

ρ	The density of the air the aircraft travels through,
v	The speed of the aircraft relative to the air,
A_D	The reference area of the aircraft experiencing drag,
C_D	The drag coefficient.

The drag forces exerted on the aircraft in each direction on the body axes can be presented as,

$$\begin{aligned} D_X &= \frac{1}{2}\rho V_{BW_X}^2 A_D C_D \\ D_Y &= \frac{1}{2}\rho V_{BW_Y}^2 A_D C_D \\ D_Z &= \frac{1}{2}\rho V_{BW_Z}^2 A_D C_D \end{aligned} \tag{2.10}$$

where, V_{BW} indicates the velocity component of the aircraft in the body axes relative to the air and D_Z is the drag experienced in the z-axis, more commonly known as Lift.

Equation 2.10 completely describe the aerodynamics of the aircraft. The drag coefficient and reference area of the quad-rotor where determined by Peddle [6] in research prior to this project, with A_D and C_D calculated as 0.5 m^2 and 1.0 respectively.

2.5.2 Actuators

The forces and moments necessary to control a quad-rotor are generated by using different speed variations of the four motors. The trust generated by each of the four motors are shown in Figure 2.6, where d represents the distance of the motor from the centre of the aircraft's body, r_D is the distance from the motor shaft to the point where the drag force is exerted of each rotor, and T_1 to T_4 are the thrust generated by each motor.

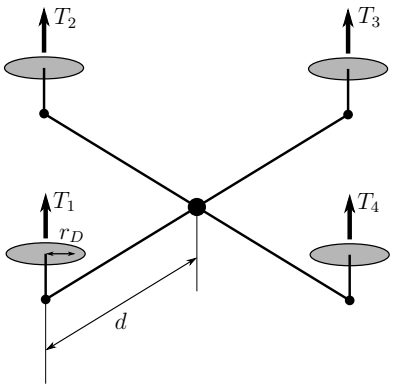


Figure 2.6: The thrust dynamics of the aircraft.

The trust generated by the motors can now be defined as virtual actuators, which is done to simplify the aircraft model. These virtual actuators are,

$$\begin{aligned} \delta_T &= T_1 + T_2 + T_3 + T_4 \\ \delta_A &= T_4 - T_2 \\ \delta_E &= T_1 - T_3 \\ \delta_R &= -T_1 + T_2 - T_3 + T_4 \end{aligned} \tag{2.11}$$

where, T_1 to T_4 represents the thrust generated by each motor respectively. δ_T is the virtual actuator that controls the upwards movement of the aircraft relative to its body axes, δ_A is the virtual aileron actuator that controls the roll of the aircraft, δ_E is the virtual elevator actuator that controls the pitch of the aircraft and δ_R is the virtual rudder actuator that controls the yaw of the aircraft.

It is important to note that the motors do not instantaneously respond to new thrust commands. A first order lag model is therefore used to capture the delay experienced by each motor and is given by

$$\dot{T} = (-T + T_R)/\tau, \tag{2.12}$$

where, T_R is the reference thrust command and τ is the motor lag time constant. The lag dynamics can then be rewritten in virtual actuator form as,

$$\dot{\delta} = (-\delta + \delta_R)/\tau. \tag{2.13}$$

The forces exerted on the aircraft and moments around the body axes due to the thrust generated by the rotors, can now be presented. The vertical force with respect to the body axes due to the rotors is given by

$$Z^T = -\delta_T, \tag{2.14}$$

the rolling moment due to the rotors is given by

$$L^T = d\delta_A, \tag{2.15}$$

the rolling moment due to the rotors is given by

$$M^T = d\delta_E, \tag{2.16}$$

and the yawing moment due to the rotors is given by

$$N^T = r_D\delta_T/R_{LD}, \tag{2.17}$$

where R_{LD} is the lift to drag ratio of the of the rotors.

Equation 2.14 to 2.17 fully describe the forces and moments generated by each rotor, which are represented as virtual actuators. The time constant τ is calculated as 0.125 seconds and the lift to drag ratio R_{LD} is given as 10.

2.5.3 Gravity

Gravity can be modelled as a single downwards force equivalent to the mass of the aircraft. The gravitational force in the inertial axis system points downwards through the centre of mass of the aircraft and is expressed by

$${F_E}^G = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix}. \tag{2.18}$$

Transforming the gravitational force to the body axis system using the DCM matrix in Equation 2.8 yields

$$\begin{bmatrix} X^G \\ Y^G \\ Z^G \end{bmatrix} = DCM \cdot \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} = \begin{bmatrix} -\sin\theta \\ \cos\theta\sin\phi \\ \sin\theta\cos\phi \end{bmatrix} mg, \tag{2.19}$$

and finally, gravity does not produce any moments on the aircraft, due to the gravitational vector acting through the centre of mass. Thus,

$$L^G = M^G = N^G = 0 \tag{2.20}$$

With the forces that comes form drag, actuators and gravity presented, the only force left that is experienced by the aircraft, comes from wind.

2.6 Wind Disturbance Model

Due to the quad-rotor moving in an outside environment, it is important to have a model that will accurately represent the wind affecting the quad-rotor's movement.

The wind model is a mathematical model describing constant winds and gusts acting on the aircraft and will be used for non-linear simulations. The wind model is calculated as the velocity of the wind in the inertial axis system, which can be seen in Figure 2.7.

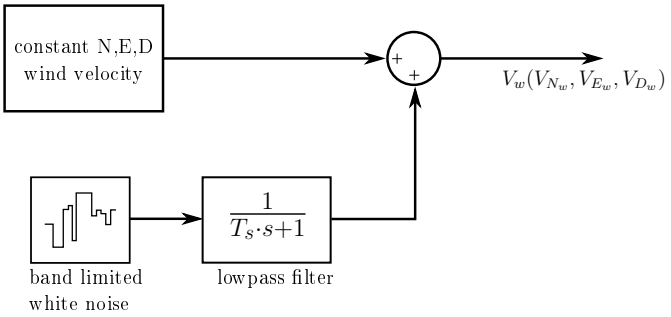


Figure 2.7: The wind disturbance model.

The constant wind is simulated as a velocity magnitude in the inertial axis system and the wind gusts are simulated as white noise passed through a low-pass filter. This is added together, and the output gives a realistic simulation of real wind velocities.

By combining the forces acting on the aircraft, the total force in each direction of the body axis system is given by

$$X = -X^G - D_X, \tag{2.21}$$

$$Y = Y_G - D_Y, \tag{2.22}$$

and

$$Z = -\delta_T - D_Z + Z^G. \tag{2.23}$$

The moment acting on the body axis is given as

$$L = d\delta_A, \tag{2.24}$$

$$M = d\delta_E, \tag{2.25}$$

and

$$N = r_D \delta_R / R_{LD}. \tag{2.26}$$

around the X_B, Y_B and Z_B axis respectively.

The forces and moments equations and the equations of motion form the previous sections combined form the full aircraft model, that is able to represent a realistic aircraft in simulation. With these equations established, the next part is to design a working control system.

2.7 Control System

The controllers used for this project were previously designed and practically verified by Peddle for the quad-rotor SLADe [6]. The in depth design and analysis of controllers are beyond the scope of this project, therefore only a brief overview of the pre-existing controllers will be discussed. In this section a overview of the aircraft model will be presented followed by an brief discussion on the pre-existing flight control system.

2.7.1 Aircraft Model

With all the components derived that represents an aircraft mathematically form the previous section, the full aircraft model is now presented in Figure 2.8.

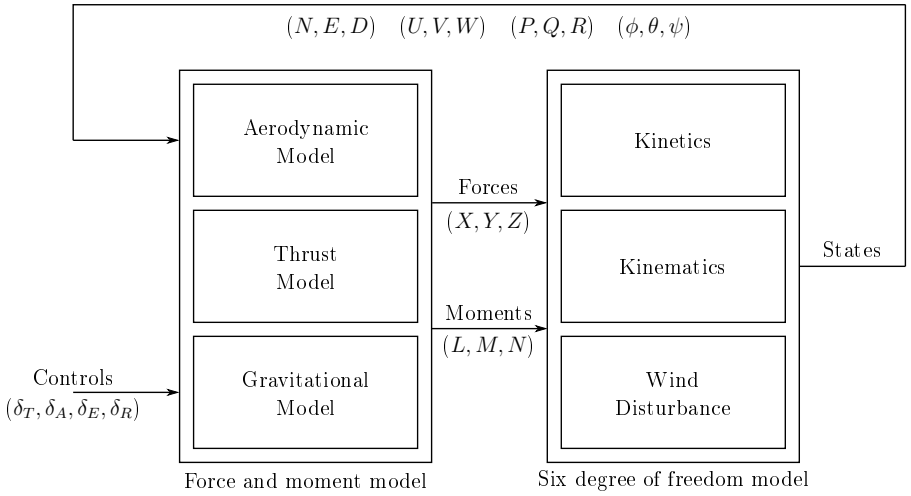


Figure 2.8: A block diagram illustration of the aircraft model adapted from Engelbrecht [2].

The block on the right represents the six degree of freedom equations that models the dynamic of the aircraft and the block on the left is all the forces acting on the aircraft. The states of the aircraft is fed back to the force and moments block, which determines the future state of the forces. The four actuator control inputs, δ_T, δ_A, δ_E and δ_R are used by the control system to make the system behave as desired.

The next subsection will discuss the pre-existing control system designed to control the aircraft model shown in Figure 2.8.

2.7.2 Pre-Existing Controllers

For the pre-existing control system design, it is assumed that the full state vector is available for feedback. A successive loop closure method is adopted instead of a full state feedback approach. The successive loop closure method is a far more practical approach and tends to render more robust and easy to use control systems compared to full-sate feedback. A top-level block diagram of this control system is shown in Figure 2.9. The flight control system of the

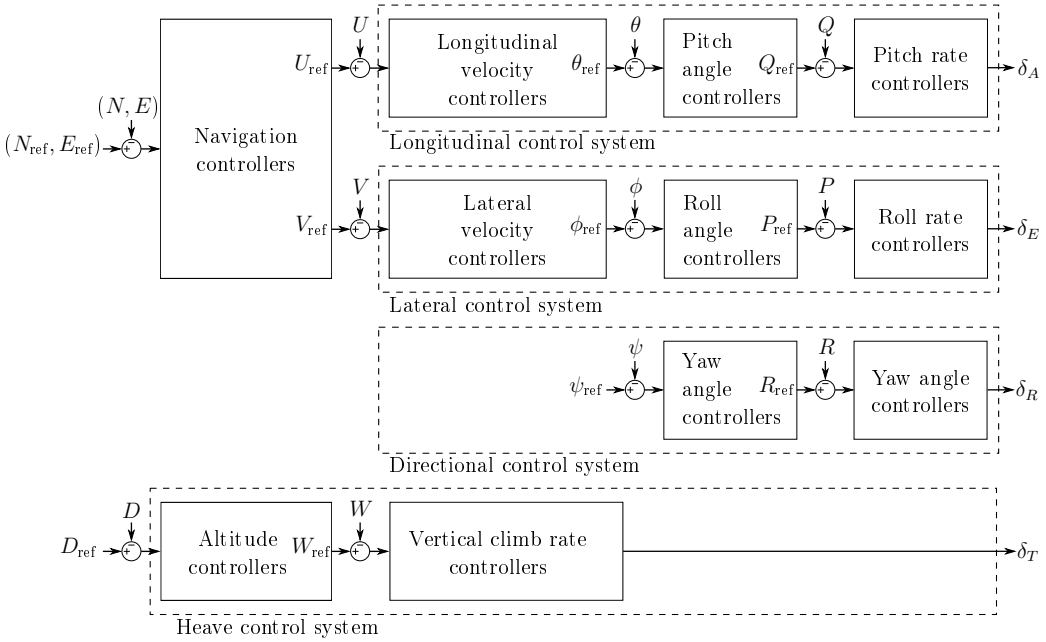


Figure 2.9: Block diagram of the pre-existing flight control system.

aircraft is decoupled into five different subsystems which are responsible for controlling the different axis of the aircraft as well as the aircraft's orientation and position in the inertial axes system. These subsystems, as seen in Figure 2.9, are called the longitudinal, lateral, directional, Heave and Navigation control systems.

The inner-most loop of each subsystem, the P, Q, R controllers, make use of a proportional-integral (PI) control approach. All the outer loops make use of proportional control (P), which eventually controls the position of the aircraft in the inertial axis system (North, East and Down). The output of the control system is the virtual actuator thrust commands described in Section 2.5.2.

With these controllers in place, it is possible to regulate the three dimensional position of the aircraft. It is also possible to operate the aircraft in different modes (e.g. velocity control or angle rate control), by manually providing input references to the inner loop controllers. The performance of the control

system is tested in Chapter 4 which showed that it is needed to improve the position tracking error and the necessary improvements are then made.

The next chapter will use the quad-rotor model as reference to design requirements, which is then used to find and develop a motion planning algorithm that is able to plan an executable trajectory for the quad-rotor to fly from a starting location to any desired goal location.

Chapter 3

Motion Planning Algorithms

After the model of the testing vehicle is obtained, the following step is to find an appropriate motion planning algorithm, that will meet the requirements set out in Section 1.3. This chapter starts by restating the requirements set out in Chapter 1 and follows with an overview of different types of motion planning algorithms, after which an appropriate algorithm is chosen. This chapter then presents different adjustments that are necessary to meet the requirements. The Chapter ends by giving an in-depth discussion on the motion planning approach that will be implemented as well as the adjustments made to meet the requirements.

3.1 Requirements

Motion planning, in the case of a rotary-wing UAV, aims to find the sequence of vehicle states that will move the aircraft from its initial state to a goal state without colliding with obstacles in the environment [7].

The motion planning algorithm designed in this project, as mentioned in Section 1.3, should be able to find a collision free trajectory in environments where both static and dynamic obstacles exist for a rotary-wing UAV. To design such a motion planning algorithm it needs to meet the requirements:

1. With more computational power given to the algorithm, the probability of finding a trajectory between any two vehicle states, if one exists, should increase.
2. The trajectories generated by the motion planning algorithm should conform to the dynamic constraints of the vehicle.
3. The final trajectory the vehicle has to follow to reach its goal should have a low probability of collision with both static and dynamic obstacles.

4. The planning algorithm should be able to find alternative trajectories when the current planned trajectory becomes conflicted.

5. All the trajectories generated by the planning algorithm should be cost-effective in terms of its length.

The next section can now use these requirements to find an appropriate motion planning algorithm for this project.

3.2 Motion Planning Approaches

This section gives an overview of the different types of motion planning approaches, after which an appropriate one is chosen, based on the requirements laid out in the previous section. The motion planning approaches discussed in this section are combinatorial algorithms, grid-based algorithms, potential fields algorithms and sampling-based algorithms which the latter is the more suitable approach for this project.

3.2.1 Combinatorial Algorithms

Combinatorial planning algorithms find trajectories through the environment without any approximations. These algorithms are referred as exact [8]. Combinatorial Algorithms are also complete, which means that for any problem instance, they will find a solution or correctly report no solution. Combinatorial algorithms make the assumption that obstacles are explicitly presented as geometric forms in the environment and use the obstacle information to find trajectories through the environment. This means that if the obstacles have certain convenient properties, such as low dimensionality, convex object models, then combinatorial algorithms will be able to find a solution relatively quickly. If obstacle representations are more complex, even just basic three dimensional geometric shapes, the problem becomes computationally expensive and, in some cases, even unattainable (refer to LaValle [8] for more information on computational complexity).

When considering a motion planning algorithm for real-time application, combinatorial techniques fall short due to their inability to find trajectories when obstacle representation becomes more complex. These types of algorithms have widely been abandoned in favour of techniques with weaker notations of completeness, but that are more efficient [8].

3.2.2 Grid-based Algorithms

Grid-based motion planning algorithms find trajectories by dividing a n dimensional environment into k^n grid cells that represent a graph. Each cell centre

is seen as a possible point of interest to which a vehicle can move. These algorithms then mark each grid cell that touches an obstacle as an invalid cell. Search algorithms like A* can be used to find a trajectory from start to goal through the cells of the graph.

As a result of the discretisation of the environment, grid-based algorithms are not complete, but rather resolution complete. This means that it will find a trajectory, if one exists, at the level of discretisation chosen. The solution trajectory found is optimal, in terms of trajectory length, for the underlying graph when using A* search. If the number of cells increase, so does the optimality of the trajectory.

The major drawback of grid-based algorithms is that it becomes unpractical for high-dimensional large spaces. The amount of memory needed to represent a grid and the time it takes to search the graph grows exponentially with each dimension added to the graph. Another drawback is that to find a trajectory between closely-spaced obstacles, the resolution of the graph needs to be high, which increases the time it takes to find a trajectory. Due to these drawbacks, grid-based algorithms do not seem to be a good choice.

3.2.3 Potential Field Algorithms

Potential field planning algorithms make use of artificial potential fields to find a collision free trajectory in the given environment. The starting position of a vehicle is seen as the position with the highest potential and the goal state the lowest. The goal exhibits an attractive field while obstacles in the environment produce repulsive fields and the vehicle will then move from the highest to lowest potential to reach its goal. By using attractive and repulsive fields, it is possible to find a collision free trajectory in a given environment.

The biggest drawback when considering potential field algorithms is that the vehicle is prone to get stuck in local minima, which occurs when all the artificial forces cancel each other out. This occur in situations where obstacles are closely spaced or an obstacle is exactly between the vehicle and goal. This means that potential field methods are not complete. These algorithms are also difficult to implement in real-life scenarios and become computationally expensive when working with environments that are higher than two dimensions [9].

Potential field algorithms, thus do not seem to meet the requirements for this thesis. Therefore a decision was made to look at sampling-based motion planning algorithms to solve the problem at hand.

3.2.4 Sampling-based Algorithms

Sampling-based motion planning algorithms explore the environment by generating random samples (vehicle states) over a configuration space, and attempt to connect these samples with collision free trajectories. A sequence of these connected samples will then form a trajectory between the start and goal. This randomised approach is able to provide solutions quickly, but does not guarantee in finding a solution if one exists. They ensure a weaker notion for completeness called, probabilistic completeness, which means that with enough samples, the probability that a solution will be found converges to one [10].

Sampling-based planners were proposed to overcome the complexity of deterministic planning algorithms for robots with six degrees of freedom, by using random computations to solve otherwise difficult problems [3]. The two most commonly-used sampling-based algorithms are probabilistic roadmap (PRM) methods and rapidly-exploring random trees (RRT) [3]. Both these algorithms, and variants thereof, are widely adopted in the field of robotics due to their simplicity and adaptability to different kind of problems.

Although sampling-based algorithms have weaker notion for completeness, their simplicity and ability to provide fast solutions are ideal for planning in dynamic environments. The sampling-based approach was therefore chosen for this project. The PRM and RRT algorithms are both investigated further in the upcoming section.

3.3 Sampling-based Motion Planning

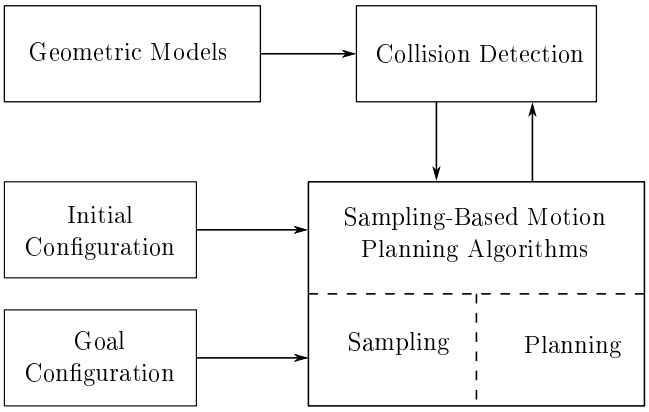


Figure 3.1: A general sampling-based planner.

Sampling-based motion planning, as mentioned previously, explores the environment by randomly sampling vehicles states and trying to connect these samples by making use of a collision detection system. Collision detection is

seen as a black box, which separates the motion planning from the geometric and kinematic model of the environment, shown in Figure 3.1. The planner is then able use the collision detection system to return collision free trajectory between two samples. A sequence of collision free trajectories between samples that connects the initial and goal location, is then considered the final solution.

Most sampling-based motion planning algorithms is closely based on the general approach shown in Figure 3.1. This following parts of this section will introduce the key concepts needed to define the sampling motion planning problem followed by the investigation of the two categories of sampling-based planning (PRM and RRT). The section will then investigate different improvements that can be made to sampling-based planning and end off by introducing the planning algorithm implemented for this project.

3.3.1 Problem Definition

In order to understand these sampling-based algorithms, some concepts must first be introduced. Most sampling-based planners operate in the configuration space (C-space), which is all the possible states a vehicle can achieve in any given environment. Free space (C_{free}) and obstacle space (C_{obs}), are the two sub-regions the configuration space consist of. The vehicle is represented by a state, q, at any instance inside C-space which is commonly referred to as a node or sample. A sequence of connected nodes is called a trajectory, P, and is considered collision free when all the trajectories between states that make up the final trajectory, lies in C_{free} and does not intersect C_{obs}. $\mathbf{q}_{\text{start}}$ is the starting state of the vehicle and $\mathbf{q}_{\text{goal}}$ is the goal state. The motion planning problem is then to find a connection between $\mathbf{q}_{\text{start}}$ and $\mathbf{q}_{\text{goal}}$ with a sequence of states inside C_{free}.

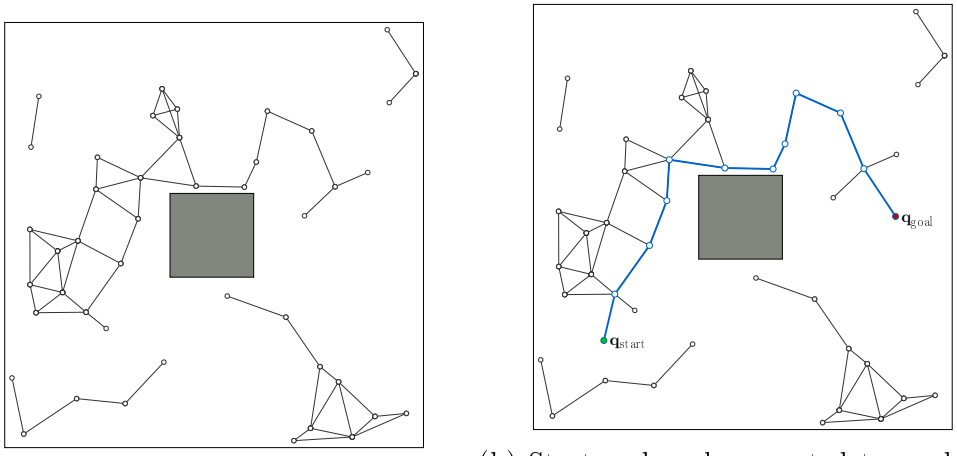
With these concepts established, the PRM and RRT algorithms can now be presented. They are the two main categories used in sampling-based motion planning and most new planners introduced are largely based on them.

3.3.2 Probabilistic Roadmap (PRM)

The probabilistic roadmap (PRM), was first introduced by Kavraki *et al.* in 1996 [11]. Since then it has been widely adopted in the field of robotics. The classic PRM starts by building a roadmap of the C-space, which is called the learning phase. This is done by sampling a random node, $\mathbf{q}_{\text{rand}}$, from the C-space. $\mathbf{q}_{\text{rand}}$ is discarded if it lies in C_{obs}, otherwise $\mathbf{q}_{\text{rand}}$ is added to the roadmap. The PRM algorithm then finds all the nodes within a specific range to $\mathbf{q}_{\text{rand}}$ and attempts to connect each neighbouring node to $\mathbf{q}_{\text{rand}}$ by using a local planner. Local planning is the process of trying to establish a connection between two nodes (it is intuitive to connect these states with a straight line

trajectory, but for most vehicles it is not feasible due to dynamic and kinematic constraints). This process is then repeated until a certain number of nodes have been sampled.

A typical roadmap, where the local planner make use of straight-line connections, is shown in Figure 3.2a. The next phase, called the query phase, of the PRM algorithm is to connect the start and goal nodes to the roadmap. A search algorithm is then employed to find the shortest trajectory between $\mathbf{q}_{\text{start}}$ and $\mathbf{q}_{\text{goal}}$ which can be seen in Figure 3.2b.



(a) Roadmap built in learning phase.

(b) Start and goal connected to roadmap in blue.

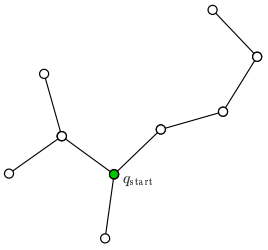
Figure 3.2: Probabilistic roadmap learning and query phase made by Elbanhawi [3], where the blue line is the trajectory found between the start and goal.

As a result of generating a roadmap and specifying the start and goal states in a subsequent phase, the PRM is able to solve different problems in the same environment. This is referred to as a multi-query planner. PRM algorithms invest most of the planning time into sampling and generating a roadmap so that the query phase can be solved quickly. The PRM, as most sampling-based planners, was shown to be probabilistic complete [12].

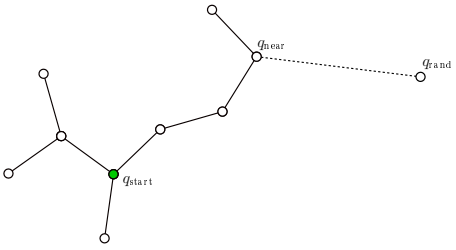
3.3.3 Rapidly-exploring Random Tree (RRT)

The rapidly-exploring random tree (RRT) was introduced by LaValle in 1998 [13] and represents another category of sampling-based planners, which are referred to as single-query planners. It was designed specifically to handle vehicles with non-holonomic constraints and problems with high degrees of freedom. The RRT has been widely adopted for many different problems, which has resulted in many variants of the RRT. The RRT is also proven to be probabilistic complete [14].

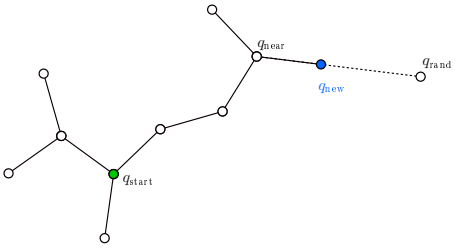
The classic RRT starts by initialising $\mathbf{q}_{\text{start}}$ in the C-space. A random sample, $\mathbf{q}_{\text{rand}}$, is then drawn from the C-space as shown in Figure 3.3b. If this sample lies in C_{obs}, it is discarded. Otherwise a nearest-neighbour search is done to find the node in the tree ($\mathbf{q}_{\text{near}}$) closest to $\mathbf{q}_{\text{rand}}$. A local planner is subsequently deployed that tries to connect $\mathbf{q}_{\text{near}}$ to $\mathbf{q}_{\text{rand}}$. If $\mathbf{q}_{\text{rand}}$ is too far from $\mathbf{q}_{\text{near}}$, the planner will return $\mathbf{q}_{\text{new}}$ and try to connect it to $\mathbf{q}_{\text{near}}$, as shown in Figure 3.3c. If the trajectory between $\mathbf{q}_{\text{near}}$ and $\mathbf{q}_{\text{new}}$ is collision free, $\mathbf{q}_{\text{new}}$ will be added to the tree, as shown in Figure 3.3d. $\mathbf{q}_{\text{near}}$ is now called the parent of $\mathbf{q}_{\text{new}}$. A new sample is then drawn and the process restarts. The algorithm terminates when $\mathbf{q}_{\text{new}}$ equals $\mathbf{q}_{\text{goal}}$, which means a collision free trajectory from start to goal has been found. This trajectory includes all the parent nodes of $\mathbf{q}_{\text{goal}}$ until it reaches $\mathbf{q}_{\text{start}}$, as seen in Figure 3.4.



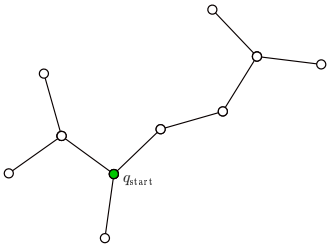
(a) The initial tree before a new sample is drawn.



(b) A new sample, $\mathbf{q}_{\text{rand}}$ is drawn, and the nearest node, $\mathbf{q}_{\text{near}}$, is found.



(c) $\mathbf{q}_{\text{new}}$ is calculated and added to the tree.



(d) The new tree after the node is added.

Figure 3.3: An illustration of the procedure of adding a new sample to the tree for the RRT.

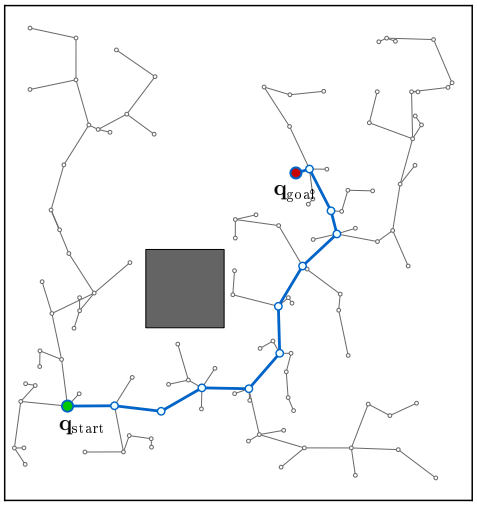


Figure 3.4: The output given by the RRT algorithm with 120 samples where the solution trajectory is shown in blue.

When both the PRM and RRT in their classic forms are examined, it is clear that there are multiple ways to improve these planners to fulfil one's needs. The key aspects where improvement can be made, are the way samples are taken from the environment, what type of local planner is used to connect nodes, how to guide the exploration to search more effectively, whether it is possible to change the sampling space to improve the speed of finding a solution and how to effectively determine the cost of connecting nodes. The next section will look at previous research done regarding these aspects and conclude with what will be needed to meet the project's requirements.

3.3.4 Modifications

A significant amount of research in sampling-based planning went into modifying the planners by changing the aspects mentioned above. The aim of these improvements are either to reduce the runtime of these algorithms, to decrease the cost of the solution or to find solutions that conform to the dynamic constraints of a vehicle. In this subsection, a number of these aspects and how they improve the sampling-based algorithms for a specific problem will be discussed.

3.3.4.1 Sampling

Sampling is the process through which all sampling-based planners are able to explore the environment. The classic implementations of the PRM and RRT uses uniform sampling. This results in the planner having a higher probability to explore wide open regions instead of narrow spaces between objects. Over the years, many strategies have been investigated to overcome the shortcom-

ings of uniformed sampling. One such strategy is boundary sampling, which forces sampling towards the boundary of obstacles, instead of free space [15]. Another strategy is called goal biasing, which attempts to connect the goal to the current tree every N^{th} iteration to force the tree to grow towards the goal region. Research has shown that sampling has little to no effect on the performance of the planner, but can be useful for certain types of problems [3]. In this project, goal biasing was implemented as re-selecting the goal state will allow the planner to find new solutions, when a dynamic obstacle moves in the way of the current trajectory towards the goal, which will be discussed in Section 3.4.3.

3.3.4.2 Optimal Planning

The ability of a sampling-based planner to find solutions in high-dimensional spaces in a reasonable amount time is one of its main advantages, but it leads to many unnecessary nodes in the final solutions. This means that most trajectories found by single-query algorithms, like the RRT, are not cost effective. Karaman and Frazzoli [16] proved that the RRT will not find an optimal solution so they then proposed a family of optimal sampling-based planners that were guaranteed to be asymptotically optimal, which means that the probability of finding an optimal solution approaches one when the number of samples approaches infinity. The planners proposed are called RRT*, PRM* and RRG* [16]. When looking back at the requirements for this project, it is clear that to find a cost effective trajectory, an optimal planning strategy will be needed.

3.3.4.3 Sampling Space

The sampling space is the part of the environment out of which nodes can be drawn. The idea behind changing the sampling space is to reduce the time it takes for a planner to find a solution or to improve the convergence rate towards an optimal solution. Most research has gone into ways of reducing the size of the sampling space. The informed RRT* [17] is an example of how changing the sampling space can improve the convergence rate of the solution. The informed RRT* retains the probabilistic completeness of the classic RRT while improving the convergence rate of the cost of the solution. When referring back to the project requirements, it will be beneficial to improve the convergence rate of the planner.

3.3.4.4 Local Planning

The local planner is used to connect sampled nodes to each other. Most planners do this by joining nodes with straight lines. In the case of UAVs with dynamic constraints, straight-line connections might not be feasible. A viable approach is to model the aircraft and sample the control space so that each node can be connected by a trajectory that represents the control inputs

needed to steer the vehicle from one node to the other. However, it must be noted that this approach depends on numerical integration, which is typically computationally expensive. Another approach to local planning is to use a predefined set of manoeuvrers that is designed to conform with the dynamic constraints of a vehicle. The local planner can then choose a combination of these manoeuvres to make a connection between nodes. Dubins path and Reeds and Shepps paths are commonly-used manoeuvres for non-holonomic vehicles [7]. They restrict the movement of the vehicles with a minimum turning radius by combining circular arcs and straight lines together. When at looking rotary-wing UAVs, it is evident that, some type of local planner that addresses the limitation of the aircraft will be needed for this project.

When looking at the PRM and RRT and all the improvements mentioned, it is not clear which one is the superior option. Due to the fact that most sampling-based planning algorithms for dynamic environments found are based on single query planners, a decision was made to use the RRT as a benchmark for this project. The reason for the use of single-query planners for dynamic environments is because these algorithms might need fewer iterations to find a new trajectory to goal when information about obstacles changes, whereas multi-query algorithms, like the PRM, might have to search the entire graph to find a new solution.

The next section will introduce motion planning for dynamic environments by firstly looking at some extensions made to the RRT that is necessary to explain the motion planning solution chosen for this project. The section ends with a more in-depth discussion on the chosen algorithm.

3.4 Dynamic Motion Planning

Planning a trajectory for a UAV from a starting to goal location when the environment is static has been thoroughly investigated over the past few years and many algorithms have proven to be very effective. However, environment in which most robots operate, has many moving obstacles and is much more difficult problem to solve. Due to these dynamic obstacles, nodes might only be safe at certain times, and therefore it is only possible to plan trajectories instead of full paths. It is also mostly impossible to predict the movement of dynamic obstacles accurately far into the future. The planning algorithm therefore has to adapt to the changing prediction about the obstacles. Therefore it was decided, to have a motion planning algorithm that will be feasible in dynamic environment, the algorithm should be able to plan, avoid and re-plan trajectories for a quad-rotor to avoid these obstacles.

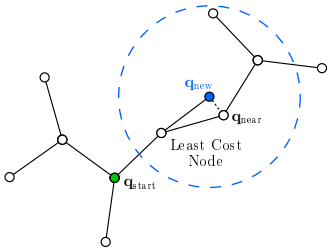
This section will investigate a solution called the RT-RRT* (Real-Time RRT*), which was initially introduced by Naderi *et al.* [18] for the use as a dynamic

planning algorithm in a simulated computer gaming environment. This algorithm is then modified to work as planning algorithm for a quad-rotor in three-dimensional space. Before the RT-RRT* is introduced, a few concepts must first be established. This section start by looking at the extensions made to the RRT called the RRT* and the informed RRT*, which is used in the RT-RRT*, before explaining this algorithm in full.

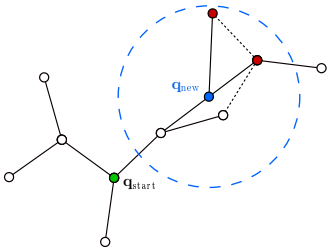
3.4.1 RRT*

The RRT* was first introduced by Karaman and Frazzoli [16] to improve the quality of trajectories generated by the classic RRT, by rewiring the connections between nodes so that the cost to move from the starting state to any node in the tree approaches optimality when the number of samples approaches infinity.

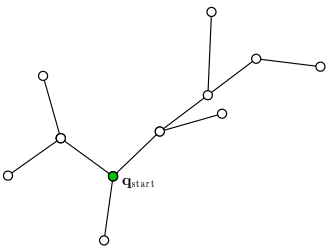
The RRT* starts in the same way as the classic RRT by initialising a starting node, $\mathbf{q}_{\text{start}}$, as the base of the tree. It then takes a random sample, $\mathbf{q}_{\text{rand}}$, from the C-space and extend it the same way as the RRT to $\mathbf{q}_{\text{new}}$. Instead of trying to make a connection between $\mathbf{q}_{\text{new}}$ and the closest node in the tree, the RRT* tries to connect $\mathbf{q}_{\text{new}}$ to the neighbouring node (neighbouring nodes are all nodes in the tree that are a certain distance away from $\mathbf{q}_{\text{new}}$) that has the lowest cost, as seen in Figure 3.5a. The cost of a node is dependent on the total travel distance to reach a node from $\mathbf{q}_{\text{start}}$. The RRT* also looks at all the neighbour nodes and if one of these nodes can connect to $\mathbf{q}_{\text{new}}$ so that the cost to reach will decrease, it will remove its previous connection and change it to the lower cost connection, which is shown in Figure 3.5b and 3.5c. This process is then repeated until a maximum number of samples have been drawn.



(a) The tree after $\mathbf{q}_{\text{new}}$ is connected to the lowest cost parent, with the rewiring region in blue.



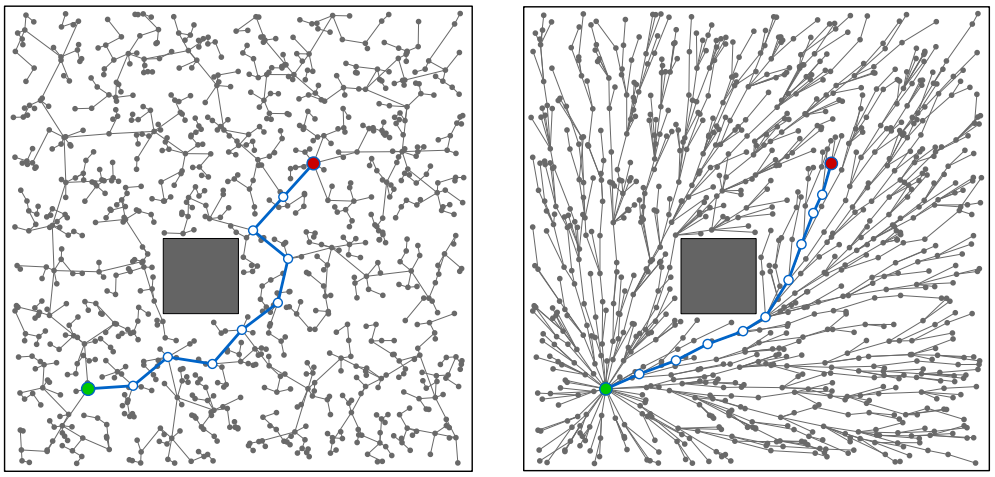
(b) The tree after all possible nodes have been rewired through $\mathbf{q}_{\text{new}}$ to reduce cost.



(c) The new tree after a node was added and rewired.

Figure 3.5: A illustration of the procedure of adding a new sample and rewiring old samples for the RRT*

The RRT* is able to decrease the cost to reach of each of the branches as more samples are added. When looking at Figure 3.6a, the trajectory found when running the RRT for 1000 samples, and Figure 3.6b, the trajectory found when running the RRT* for 1000 samples, it is clear that the total trajectory length when using the RRT* is much shorter than the one found when using the RRT for the same setup.



(a) RRT (b) RRT*

Figure 3.6: The comparison of trajectory length between the RRT and the RRT* algorithm when 1000 has been added to the tree.

The disadvantage of the RRT* is that it takes much longer for the planner to explore the C-space. The extra computations of the RRT* influence the time it takes to find an initial solution with little optimization. This means that for the RRT* is more expensive than the RRT. The rate at which trajectories become more optimal is called the convergence rate. The informed RRT* (discussed below) is a way to improve the convergence rate of the RRT* so that the time it takes to find more optimal trajectories is reduced.

3.4.2 Informed RRT*

The informed RRT* was introduced by Gammell *et al.* [17] to improve the convergence rate of the RRT* while still retaining the same probabilistic completeness of the RRT. The informed RRT* does this by reducing the sampling space to an ellipsoidal subset after an initial solution has been found, to focus on optimising the trajectory from start to goal. The ellipsoidal subset is constructed in such a way to guarantee that all samples outside the subset will not improve the quality of the solution. The ellipsoidal subset is constructed, as shown in Figure 3.7, by taking the start, $\mathbf{q}_{\text{start}}$, and goal, $\mathbf{q}_{\text{goal}}$, nodes as the focal points of the ellipse. C_{min} is the theoretical minimum cost between the

start and goal nodes and C_{best} is the cost of the best solution found for the current iteration.

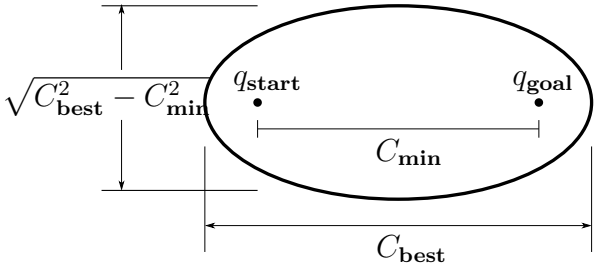
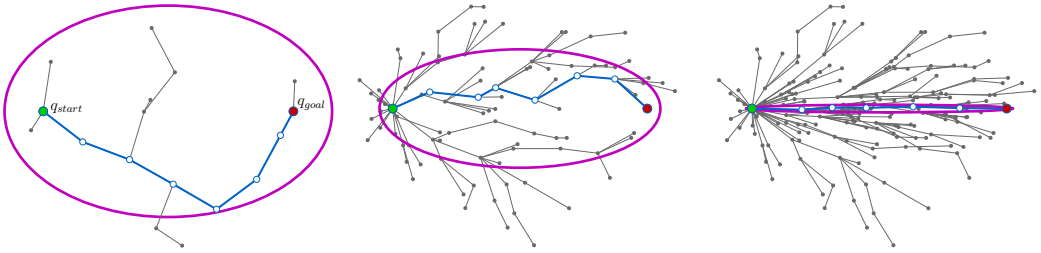


Figure 3.7: The shape of the ellipse constructed for the Informed RRT*

The informed RRT* behaves the same way as the RRT* until a first solution has been found. It then proceeds to construct the ellipsoidal subset by using the cost of the current solution, after which it will only sample from this subset. If a lower-cost solution is found, the informed RRT* will reconstruct the subset and reduce the sampling space even further. In the case where there are no obstacles in the environment, the solution will converge towards the optimal solution (for example, a straight line between $\mathbf{q}_{\text{start}}$ and $\mathbf{q}_{\text{goal}}$) in finite time, shown in Figure 3.8.

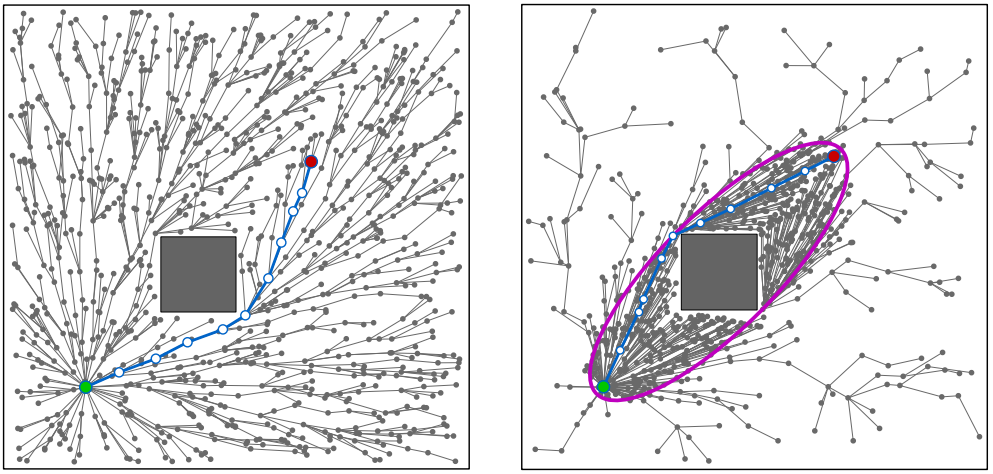


(a) (b) (c)

Figure 3.8: A visual representation of how the informed RRT* changes its sampling space as the cost of the solution (in blue) reduces.

When comparing the RRT* and informed RRT* in the same environment, shown in Figure 3.9, it is clear that the Informed RRT* is able to find better quality solutions for the same number of samples. The figure also shows how the informed RRT* initially sampled the whole configuration space and how the sample space changed when a solution was found.

The main challenge for the Informed RRT*, in an environment with dynamic obstacles, is that the sampling space changes. If obstacles move in such a way that it will cause a collision, and the informed RRT* has reduced the sampling space too much, the planning algorithm will not be able to use the rest of the environment to find a collision free solution, because it is limited to



(a) RRT* (b) Informed RRT*

Figure 3.9: A comparison of the trees (in grey) and resultant final trajectories (in blue) generated by the RRT* and the Informed RRT* when 1000 nodes has been added to the tree.

only sampling in the ellipsoidal subset. This might result in the planner not able to find a solution at all (for example, if the ellipsoidal subset is smaller in diameter than the obstacle). This issue can be avoided by splitting the sampling between the ellipsoidal subset and the whole configuration space.

With the RRT* and informed RRT* investigated, it is now possible to introduce the real-time RRT*.

3.4.3 RT-RRT*

The RT-RRT* was designed to be used as a planning algorithm for simulated gaming environments [18], but due to the way RRT-based planners work, it can easily be adjusted to solve many different kinds of problems. The version of the RT-RRT* used in this project will make a few changes to the original form so that it can be used as a motion planning solution for the problem at hand. The classic RT-RRT* is discussed in this section and all changes made will be presented in Chapter 5, where it will be implemented and tested. This Section starts by explaining the main loop of the RT-RRT*, followed by the inner loops that make up the rest of the algorithm.

The main loop of the RT-RRT* is shown in Algorithm 1. It starts by initializing the tree with a starting position, $\mathbf{q}_{\text{start}}$, which is the origin of the tree and it is where the agent, $\mathbf{q}_{\text{a}}$, will begin its movement. The two different queues used for rewiring, Q_{r} and Q_{s}, in Algorithm 2 is also initialised here. After the initialisation step, the algorithm updates the position of the agent and the dynamic obstacles before starting the sampling and expansion process. The

Algorithm 1: Real-time RRT*

```
   Input  : q_a, q_start, q_goal, X_obs
1  N.initialize(q_a, q_goal, Q_r, Q_s)
2  while q_a has not reached q_goal do
3  |  Update q_a, X_obs
4  |  while time left for expansion and rewiring do
5  |  |  Expand and rewire the nodes in N using Algorithm 2 and X_obs
6  |  end
7  |  Search for a trajectory from q_start to q_goal (q_start, q_1, ..., q_goal)
8  |  if q_a is close to q_start then
9  |  |  q_start ← q_1
10 |  end
11 |  Move the agent towards q_start for time τ
12 end
```

algorithm then expands and rewires the tree from the current root, $\mathbf{q}_{\text{start}}$, for a limited time (lines 4-6). When the expansion and rewiring time has expired, the RT-RRT* proceeds to search for a trajectory from $\mathbf{q}_{\text{start}}$ to $\mathbf{q}_{\text{goal}}$ from the samples. If a trajectory is found, $\mathbf{q}_{\text{start}}$ moves to the next node in the trajectory, and the agent starts its movement. If no solution ins found, the algorithm will repeat line 3-7 until a solution is found and the agent will stay at the starting position until then. When the agent is close to $\mathbf{q}_{\text{start}}$, it will move $\mathbf{q}_{\text{start}}$ to the next node in the planned trajectory (lines 8-10). This means that the agent moves along the planned trajectory towards the goal while the trajectory gets updated and improved.

Tree expansion and rewiring is introduced in Algorithm 2. Samples, $\mathbf{q}_{\text{rand}}$, are drawn from the environment (line 1) and added to the tree as $\mathbf{q}_{\text{new}}$ (line 3 and 7), which will eventually cover the whole environment. The newly sampled nodes are used to rewire random parts of the tree around itself (lines 7 and 8) and will be explained later when Algorithm 4 is introduce. This rewiring is needed because of the constantly changing position of dynamic obstacles and the changing root node. $X_{\text{neighbours}}$ (line 4), the same as with RRT*, are all the neighbouring nodes around $\mathbf{q}_{\text{new}}$ with a maximum distance of r_n away. Line 5 checks if there possible collision free connection between $\mathbf{q}_{\text{new}}$ and one of its neighbours. Lines 9 and 11 do two different rewiring strategies and will be explained later in this section.

Algorithm 2: Tree Expansion and Rewiring

Input : N, Q_r, Q_s, r_n

1 Sample $\mathbf{q}_\mathrm{rand}$ using Equation 3.1
2 $\mathbf{q}_\mathrm{closest}$ = the nearest node from the tree to $\mathbf{q}_\mathrm{rand}$
3 $\mathbf{q}_\mathrm{new}$ = the distance r_n away from $\mathbf{q}_\mathrm{closest}$ towards $\mathbf{q}_\mathrm{rand}$
4 $X_\mathrm{neighbors}$ = find all nodes in tree within radius r_n from $\mathbf{q}_\mathrm{new}$
5 **if** *collsion free trajectory between* $\mathbf{q}_{new}$ *and a neighbor is found* **then**
6 | AddNodeToTree(N, $\mathbf{q}_\mathrm{new}$, $\mathbf{q}_\mathrm{closest}$, $X_\mathrm{neighbors}$) (Algorithm 3)
7 | Push $\mathbf{q}_\mathrm{new}$ to first of Q_r
8 | RewireRandomNode(Q_r, N) (Algorithm 4)
9 **end**
10 RewireFromRoot(Q_s, N) (Algorithm 5)

The random sampling done in line 1 from Algorithm 2 uses Equation 3.1 as a sampling strategy. P_r is a random number between 0 and 1. α and β are user-given constants. α determines how often the goal is sampled to bias the search towards the goal and β determines how often a sample is drawn from the whole C-space or from the ellipsoidal subset, the same ellipsoidal subset as mentioned in Section 3.4.2 for the informed RRT*. The sampling strategy equation is,

$$\mathbf{q}_\mathrm{rand} = \begin{cases} \mathrm{Goal}(\mathbf{q}_\mathrm{goal}) & \text{if } P_r < 1-\alpha, \\ \mathrm{Uniform}(C\text{-space}) & \text{if } P_r \leq \frac{1-\alpha}{\beta} \text{ or when no trajectory has been found,} \\ \mathrm{Ellipsis}(\mathbf{q}_\mathrm{start}, \mathbf{q}_\mathrm{goal}) & \text{otherwise.} \end{cases} \tag{3.1}$$

The changing environment may cause the current chosen trajectory to goal to become conflicted, which means a new solution needs to be found. Thus, as opposed to the informed RRT*, samples are sometimes drawn outside of the ellipse. This allows the algorithm to still expand and rewire other parts of the tree which might be needed to find a collision-free trajectory. Line 4 in Algorithm 2 does the same as the RRT* and returns all nodes in the tree, N, that are at most a distance of r_n away from $\mathbf{q}_\mathrm{new}$.

When a dynamic obstacle's collision area, which is the area around the obstacle where collision will occur, overlaps with a node $\mathbf{q}_i$, the cost to reach this node becomes infinity. This leads to all the nodes that branch from $\mathbf{q}_i$ to have a cost of infinity as well. Thus, when the rewiring algorithms in line 8 and 10 from Algorithm 2 tries to connect a node to a neighbour to reduce its cost, rewiring will create a new trajectory for the nodes with a cost of infinity. Algorithm 4 and 5 explain how the rewiring of a tree is done.

Algorithm 3: Add Node To Tree

Input : N, $\mathbf{q}_{\text{new}}$, $\mathbf{q}_{\text{closest}}$, $X_{\text{neighbours}}$

1 $c_{\min} = \text{Cost}(\mathbf{q}_{\text{closest}}) + \text{Dist}(\mathbf{q}_{\text{closest}}, \mathbf{q}_{\text{new}})$
2 **for** $\mathbf{q}_i$ *in* $X_{Neighbors}$ **do**
3 $\quad c_{\text{new}} = \text{Cost}(\mathbf{q}_i) + \text{PathCost}(\mathbf{q}_i, \mathbf{q}_{\text{new}})$
4 $\quad$ **if** $c_{new} < c_{min}$ *and* $Path(\mathbf{q}_i, \mathbf{q}_{new}) \in C_{free}$ **then**
5 $\quad\quad \mathbf{q}_{\text{new}}.parent = \mathbf{q}_i$
6 $\quad\quad c_{\min} = c_{\text{new}}$
7 $\quad$ **end**
8 **end**
9 Push $\mathbf{q}_{\text{new}}$ to N if parent was found.

Algorithm 3 finds the lowest cost neighbouring nodes by using the function $\text{Cost}(\mathbf{q}_i)$ in lines 3-6. It is important to note that the cost of a node is related to the length of the trajectory from $\mathbf{q}_{\text{start}}$ to $\mathbf{q}_i$. This means that whenever a node in a trajectory towards $\mathbf{q}_i$ gets rewired, the cost of $\mathbf{q}_i$ will also change. Therefore, when $\text{Cost}(\mathbf{q}_i)$ is called, it will recompute the cost from $\mathbf{q}_{\text{start}}$ to $\mathbf{q}_i$. It should also be noted that if $\text{Cost}(\mathbf{q}_i)$ is called and one of the parents of $\mathbf{q}_i$ is blocked by a dynamic obstacle, the cost of $\mathbf{q}_i$ will be infinity.

Rewiring is done when a node ($\mathbf{q}_i$) gets a lower cost value by passing through another node instead of its current parent. For the RT-RRT*, rewiring should be done when a new node ($\mathbf{q}_{\text{new}}$) is added to the tree (same as RRT*). Rewiring should also happen around already added nodes when the root node ($\mathbf{q}_{\text{start}}$) changes and when dynamic obstacles move to a new location (as opposed to RRT*). This means that large portions of the tree need to be rewired constantly to compensate for the changes, which is done by rewiring random parts of the tree (Algorithm 4) as well as rewiring that start from the root node (Algorithm 5).

Algorithm 4: Rewire random nodes

```
   Input  : N, Q_r
 1 repeat
 2 |  q_r = PopFirst(Q_r)
 3 |  X_neighbours = FindNodeNeighbours(q_r, N)
 4 |  for q_n in X_neighbours do
 5 |  |  c_old = Cost(q_n)
 6 |  |  c_new = Cost(q_r) + PathCost(qr,q_n)
 7 |  |  if c_new < c_old and Path(qr, q_n) ∈ C_free then
 8 |  |  |  q_r.parent = q_n
 9 |  |  |  Push q_n to the end of Q_r
10 |  |  end
11 |  end
12 until time is up or Q_r is empty;
```

Lines 5-8 of Algorithm 4 and Lines 8-11 of 5 will rewire the neighbouring nodes, $X_{\text{neighbours}}$, through $\mathbf{q}_r$ and $\mathbf{q}_s$ respectively. Algorithm 4 rewires random parts of the tree starting from nodes around $\mathbf{q}_{\text{new}}$ that are added to Q_r in Algorithm 2. Then if rewiring happens to any $\mathbf{q}_n$, Algorithm 4 adds $\mathbf{q}_n$ to Q_r since the nodes around $\mathbf{q}_n$ have the potential to be rewired (line 9). Algorithm 5 focusses rewiring at the root of the tree, $\mathbf{q}_{\text{start}}$, and pushes all its Neighbours to Q_s. It then continues to push nodes to Q_s with greater distance from $\mathbf{q}_{\text{start}}$ by popping nodes, $\mathbf{q}_s$ (line 5), and pushing $\mathbf{q}_n$ around $\mathbf{q}_s$ (line 14) when the condition from line 13 is met. Rewiring continues at each iteration until the condition from line 12 of Algorithm 4 and 17 of 5 is met. If the time for rewiring is up and there are still nodes in Q_r and Q_s, it will continue rewiring them in the next iteration.

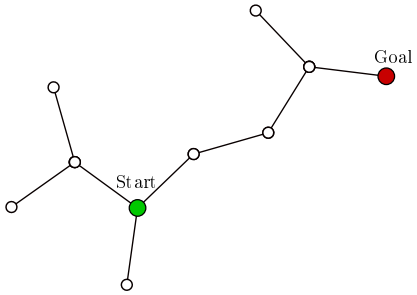
Algorithm 5: Rewire from root of tree

```
   Input  : N, Q_s
 1 if Q_s is empty then
 2 |  Push q_start to Q_s
 3 end
 4 repeat
 5 |  q_s = PopFirst(Q_s)
 6 |  X_neighbors = FindNodeNeighbors(q_s, N)
 7 |  for q_n in X_neighbours do
 8 |  |  c_old = Cost(q_n)
 9 |  |  c_new = Cost(q_s) + PathCost(q_s,q_n)
10 |  |  if c_new < c_old and Path(q_s, q_n) ∈ C_free then
11 |  |  |  q_s.parent = q_n
12 |  |  end
13 |  |  if q_n has not been pushed to Q_s after restarting Q_s then
14 |  |  |  Push q_n to the end of Q_s
15 |  |  end
16 |  end
17 until time is up or Q_s is empty;
```

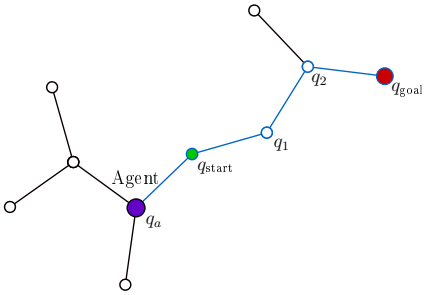
By combining the random rewiring of Q_r and the focussed sampling of the informed-RRT* ellipse, it is possible to quickly rewire the trajectory from $\mathbf{q}_{\text{start}}$ to $\mathbf{q}_{\text{goal}}$, which will yield a more optimal solution. On the other hand, using Q_s and Algorithm 5, the trajectory to each node in the tree gets more optimal, starting from the root and moving outwards. Since Algorithm 5 only rewires the already expanded tree at each iteration, it needs to be restarted when the environment changes drastically. Therefore Q_s gets restarted when the root node, $\mathbf{q}_{\text{start}}$, changes or a dynamic obstacle moves in the region where Algorithm 5 has already rewired. Restarting Q_s means that the rewiring has to start form $\mathbf{q}_{\text{start}}$ again (line 13). It should be noted that only the current locations of obstacles are used when expansion and rewiring are done, therefore it is ideal if the planning algorithm is able to quickly perform rewiring and update the current solution.

Figure 3.10 shows the full RT-RRT* algorithm visually. Figure 3.10a is an example tree just after expansion and rewiring has been done for the current iteration. The algorithm then tries to find a trajectory from $\mathbf{q}_{\text{start}}$ to $\mathbf{q}_{\text{goal}}$ (Algorithm 1, line 7) before restarting the process. At the next iteration, if a trajectory is found, the root node is moved towards the next node in the trajectory and the agent then starts its movement, shown in Figure 3.10b. The RT-RRT* does the expansion and rewiring alongside the movement of the agent and tries to find a better quality trajectory to execute, shown in Figure 3.10c. If the Agent gets close to $\mathbf{q}_{\text{start}}$, $\mathbf{q}_{\text{start}}$ will move to the next node

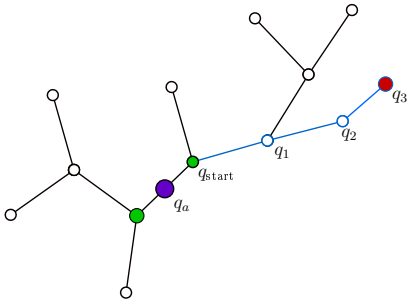
in the trajectory and Q_s will be restarted, shown in Figure 3.10d. This process is repeated until the agent has reached its goal.



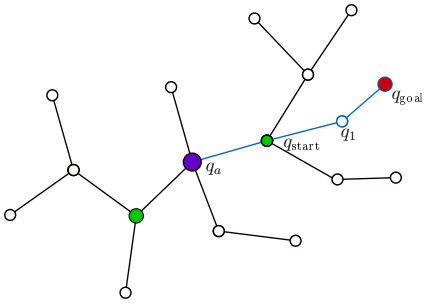
(a) The tree after a trajectory has been found between the start and goal nodes.



(b) $\mathbf{q}_{\text{start}}$ is moved to the next node in the trajectory towards the goal and the agent is initialized.



(c) The agent moves along the trajectory towards $\mathbf{q}_{\text{start}}$ while the planner updates and improve the trajectory between $\mathbf{q}_{\text{start}}$ and $\mathbf{q}_{\text{goal}}$.



(d) The agent reaches $\mathbf{q}_{\text{start}}$ and $\mathbf{q}_{\text{start}}$ now moves towards the next node.

Figure 3.10: A illustration of the expansion and rewiring procedure of the RT-RRT* alongside a moving agent.

Now that the motion planning algorithm has been identified and explained, the next step in planning a trajectory for a quad-rotor with dynamic constraints, is to find a suitable way to connect nodes together so that it is possible for the quad-rotor to execute. This means that a local planner needs to be designed that will conform to the dynamic constraints of the quad-rotor. The next chapter will look at two different ways to connect nodes and decide on which will work for this project. Chapter 5 will then take the local planner, combine it with the RT-RRT* algorithm and run simulations to test how well the solution works for this type of problem.

Chapter 4

Local Planning Method

The previous chapter introduced ways to explore the environment to plan a trajectory from a starting to a goal location, but it did not take into consideration the fact that any real-world aircraft is limited in the way it can move, due to physical and dynamic constraints. When planning trajectories for UAVs, like the quad-rotor used in this project, it is necessary to take these constraints into account. This is usually done by employing a local planning method that is able to connect nodes in such a way that it is more suitable for the specific vehicle to execute.

The chapter will explore two different local planning methods. The first method will look at an optimal way to connect two nodes, by using a linear state space representation of a vehicle. This method will then be compared to a manoeuvre based local planner, which uses a set of geometric-based manoeuvre that conforms to the vehicle constraints, to make a connection between nodes.

4.1 Optimal Control Local Planning

The optimal local planning discussed in this section is based on the work done by Webb and van den Berg in 2012 [19]. They introduced an optimal local planning technique for systems with linear differential constraints. Their approach is able to guarantee asymptotically optimal connections between any two nodes for any system with controllable linear dynamics in state spaces with any number of dimensions.

This section investigates their method and implements it for a simple problem to see how viable it is for dynamic planning. The section starts by defining the optimal trajectory planning problem. It then moves on to explain the formulation for a fixed final state and fixed final time optimal control problem and then extends it by deriving a solution for a fixed final state free final time

problem. This section then ends of by implementing the method for a simple car with a 4-D state space and double integral dynamics problem and running timing information to measure the performance and show if the specific method is a viable option for this project.

4.1.1 Problem Definition

Let $\mathbf{X} = \mathbb{R}^n$ and $\mathbf{U} = \mathbb{R}^m$ be the the state space and control input space of the vehicle respectively. The dynamics of a vehicle can be defined by the following linear system equation,

$$\dot{\boldsymbol{x}}[t] = \mathbf{A}\mathbf{x}[t] + \mathbf{B}\boldsymbol{u}[t] + \boldsymbol{c}, \tag{4.1}$$

where $\boldsymbol{x}[t] \in \mathbf{X}$ is the states and $\boldsymbol{u}[t] \in \mathbf{U}$ is the control inputs of the vehicle. $\mathbf{A} \in \mathbb{R}^{n\times n}$, $\mathbf{B} \in \mathbb{R}^{n\times m}$ and $\boldsymbol{c} \in \mathbb{R}^n$ are constants that define the vehicle model.

The trajectory between any two states is defined as a tuple $\pi = (\boldsymbol{x}, \boldsymbol{u}, \tau)$ where τ is the time it takes the given vehicle to complete a trajectory. The cost $c[\pi]$ of a trajectory, π, is defined as,

$$c[\pi] = \int_0^\tau (1 + \boldsymbol{u}[t]^T \mathbf{R}\boldsymbol{u}[t])\ dt, \tag{4.2}$$

which penalises both the time it takes to complete a trajectory as well as the total control input necessary to reach the goal. $\mathbf{R} \in \mathbb{R}^{m\times m}$ is a positive definite constant matrix that weighs the cost of the control inputs.

The problem for this section is then, given any two states $\boldsymbol{x}_0 \in \mathbf{X}$ and $\boldsymbol{x}_1 \in \mathbf{X}$, in finding a trajectory, $\pi[\boldsymbol{x}_0, \boldsymbol{x}_1]$, that connects these two states, with minimal cost.

4.1.2 Optimal State Connection

This section discusses how to compute the optimal trajectory and minimum cost between any two state, $\boldsymbol{x}_0 \in \mathbf{X}$ and $\boldsymbol{x}_1 \in \mathbf{X}$. First the optimal control policy in the case where a fixed arrival time is given is discussed. This analysis is then extended to find the optimal arrival time and it end off by showing how to calculate the optimal trajectory between the given states.

4.1.2.1 Fixed Final State and Fixed Final Time

Given a final arrival time, τ, and the states $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, a trajectory should be found such that $\boldsymbol{x}[0] = \boldsymbol{x}_0$, $\boldsymbol{x}[\tau] = \boldsymbol{x}_1$ and $\dot{\boldsymbol{x}}[t] = A\boldsymbol{x}[t] + B\boldsymbol{u}[t] + \boldsymbol{c}$ (for $0 \leq t \leq \tau$) minimises the cost function. Let $\mathbf{G}[t]$ be the weighted controllability Gramian and is given by

$$\mathbf{G}[t] = \int_0^t \exp[\mathbf{A}(t-t')]\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T \exp[\mathbf{A}^T(t-t')]\ dt', \tag{4.3}$$

which is the solution of the Lyapunov equation,

$$\dot{\mathbf{G}}[t] = \mathbf{A}\mathbf{G}[t] + \mathbf{G}[t]\mathbf{A}^T + \mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T, \quad \mathbf{G}[0] = 0. \tag{4.4}$$

Further, let $\bar{\boldsymbol{x}}[t]$, in Equation 4.5, be the state of the vehicle at time t, if no control input were given, starting at $\boldsymbol{x}_0$ when $t = 0$,

$$\bar{\boldsymbol{x}}[t] = \exp[\mathbf{A}t]\boldsymbol{x}_0 + \int_0^t \exp[\mathbf{A}(t - t')]\boldsymbol{c}\, dt', \tag{4.5}$$

which is the solution to the differential equation,

$$\dot{\bar{\boldsymbol{x}}}[t] = \mathbf{A}\bar{\boldsymbol{x}}[t] + \boldsymbol{c}, \qquad \bar{\boldsymbol{x}}[0] = \boldsymbol{x}_0. \tag{4.6}$$

This means that the optimal control policy for a fixed final state and fixed final time can be obtained by using,

$$\boldsymbol{u}[t] = \mathbf{R}^{-1}\mathbf{B}^T \exp[\mathbf{A}^T(\tau - t)]\mathbf{G}[\tau]^{-1}(\boldsymbol{x}_1 - \bar{\boldsymbol{x}}[\tau]). \tag{4.7}$$

With the above equation, it is now possible to derive a solution that will find the optimal arrival time for a minimum-cost trajectory.

4.1.2.2 Optimal Arrival Time

The optimal arrival time between the states $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ is denoted as τ^* and can be calculated by adding Equation 4.7 to the cost function of Equation 4.2. This yields a closed-form solution that will give the cost of the optimal trajectory between any two states for a given arrival time, shown below:

$$c[\tau] = \tau + (\boldsymbol{x}_1 - \bar{\boldsymbol{x}}[\tau])^T\mathbf{G}[\tau]^{-1}(\boldsymbol{x}_1 - \bar{\boldsymbol{x}}[\tau]), \tag{4.8}$$

The optimal arrival time (τ^*) is where the cost function is at its minimum. It is therefore possible to find τ^* by taking the derivative of the cost function with respect to τ (denoted as $\dot{c}[\tau]$) and solving $\dot{c}[\tau] = 0$ for τ. The derivative is given by

$$\dot{c}[\tau] = 1 - 2(\mathbf{A}\boldsymbol{x}_1 + \boldsymbol{c})^T\boldsymbol{d}[\tau] - \boldsymbol{d}[\tau]^T\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T\boldsymbol{d}[\tau], \tag{4.9}$$

where $\boldsymbol{d}[\tau]$ is defined as

$$\boldsymbol{d}[\tau] = \mathbf{G}[\tau]^{-1}(\boldsymbol{x}_1 - \bar{\boldsymbol{x}}[\tau]), \tag{4.10}$$

which is the answer that gives an optimal trajectory.

4.1.2.3 Optimal Trajectory

Given the optimal arrival time τ^* as defined previously, it is now possible to find the corresponding optimal trajectory $\pi^*[x_0, x_1] = (\boldsymbol{x}, \boldsymbol{u}, \tau^*)$. This can be done by inserting the control policy of Equation 4.7 into Equation 4.1. This yields the differential equation

$$\dot{\boldsymbol{x}}[t] = \mathbf{A}\boldsymbol{x}[t] + \mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T\boldsymbol{y}[t] + \boldsymbol{c}, \quad \boldsymbol{x}[\tau^*] = \boldsymbol{x}_1, \tag{4.11}$$

where $\boldsymbol{y}[t]$ is defined as

$$\boldsymbol{y}[t] = \exp[\mathbf{A}^T(\tau^* - t)]\boldsymbol{d}[\tau^*], \tag{4.12}$$

such that the control policy can be written as

$$\boldsymbol{u}[t] = \mathbf{R}^{-1}\mathbf{B}^T\boldsymbol{y}[t]. \tag{4.13}$$

It should be noted that Equation 4.12 is the solution to the differential equation

$$\dot{\boldsymbol{y}}[t] = -\mathbf{A}^T\boldsymbol{y}[t], \quad \boldsymbol{y}[\tau^*] = \boldsymbol{d}[\tau^*] \tag{4.14}$$

and by combining this with Equation 4.11 gives the differential equation

$$\begin{bmatrix}\dot{\boldsymbol{x}}[t] \\ \dot{\boldsymbol{y}}[t]\end{bmatrix} = \begin{bmatrix}\mathbf{A} & \mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T \\ 0 & -\mathbf{A}^T\end{bmatrix}\begin{bmatrix}\boldsymbol{x}[t] \\ \boldsymbol{y}[t]\end{bmatrix} + \begin{bmatrix}\boldsymbol{c} \\ \mathbf{0}\end{bmatrix}, \quad \begin{bmatrix}\boldsymbol{x}[\tau^*] \\ \boldsymbol{y}[\tau^*]\end{bmatrix} = \begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{d}[\tau^*]\end{bmatrix}. \tag{4.15}$$

Equation 4.15 has a solution that yields

$$\begin{aligned}\begin{bmatrix}\boldsymbol{x}[t] \\ \boldsymbol{y}[t]\end{bmatrix} = &\exp\left[\begin{bmatrix}\mathbf{A} & \mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T \\ 0 & -\mathbf{A}^T\end{bmatrix}(t-\tau^*)\right]\begin{bmatrix}\boldsymbol{x}_1 \\ \boldsymbol{d}[\tau^*]\end{bmatrix} + \\ &\int_{\tau^*}^{t}\exp\left[\begin{bmatrix}\mathbf{A} & \mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T \\ 0 & -\mathbf{A}^T\end{bmatrix}(t-\tau')\right]\begin{bmatrix}\boldsymbol{c} \\ \mathbf{0}\end{bmatrix}\, dt'.\end{aligned} \tag{4.16}$$

This equation provides a solution for $\boldsymbol{x}[t]$, which is the optimal trajectory $(\pi^*[x_0, x_1])$ between $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$. After solving $\boldsymbol{x}[t]$, Equation 4.7 can be used to find the control input, $\boldsymbol{u}[t]$.

4.1.2.4 Implementation

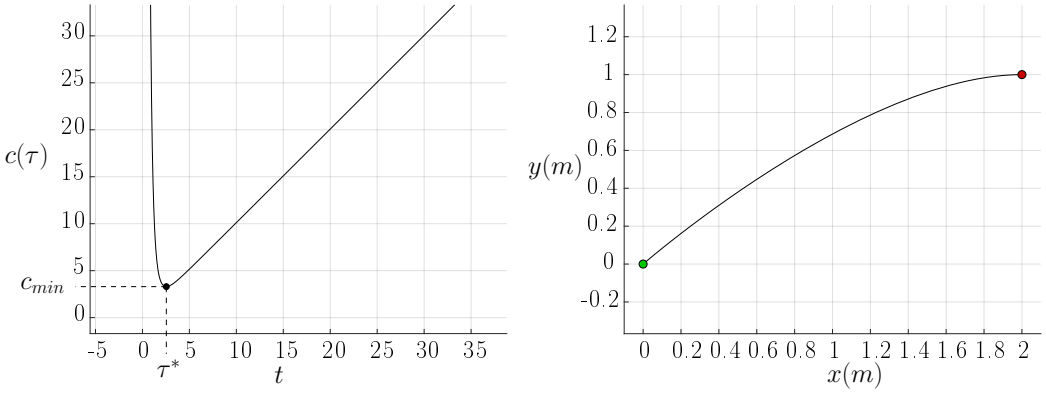
The optimal trajectory planning method was implemented for a linear double integrator model, the same example presented in [19], to determine if this method would work as local planner for the RT-RRT* motion planning algorithm. The idea is that if the optimal trajectory planning does not work for a simple model, it will not work for a more complex model. This example represents a robot that is able to move in any direction by controlling its acceleration. The states, system matrices and weighting matrix for this example are given by,

$$\boldsymbol{x} = \begin{bmatrix}x \\ y \\ \dot{x} \\ \dot{y}\end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix}0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0\end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix}0 \\ 0 \\ 1 \\ 1\end{bmatrix}, \quad \mathbf{R} = r\begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix}. \tag{4.17}$$

where x and y represent the position states in a two-dimensional environment and r is the control penalty.

To complete the specifications of the model in Equation 4.1, the control input, $\boldsymbol{u}$, is set to be the robot's acceleration, $\boldsymbol{a}$, and $\boldsymbol{c} = \boldsymbol{0}$. The control penalty, r, is chosen as 0.25 so that the time length of a trajectory contributes more to the cost than the inputs.

The optimal local planning method was implemented on the model presented above with the starting and goal states given as, $\boldsymbol{x}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^T$ and $\boldsymbol{x}_1 = \begin{bmatrix} 2 & 1 & 1 & 0 \end{bmatrix}^T$ respectively. Figure 4.1a shows the cost of the trajectory as a function of time, where $c_{\min} = 3.3$ (minimum cost) at $\tau^* = 2.544\,\text{s}$ (optimal arrival time). Figure 4.1b then shows the optimal trajectory given by equation 4.16 for the arrival time, τ^*.



(a) The cost function of the trajectories between $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$.

(b) The minimum cost trajectory between $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$.

Figure 4.1: Minimised cost function with optimal trajectory for a linear double integrator model.

The method was implemented as a local planner for the informed RRT* of Chapter 3 in the same simulation environment presented there. The integration of Equations 4.4, 4.6, and 4.15 was solved by a 4th-order Runge-Kutta numerical integration method. Even though this example has a closed form solution [19], numerical integration methods will be necessary for more complex vehicle models, and was therefore implemented to make accurate conclusions about optimal trajectory planning as a local planner.

Figure 4.2 shows the solution found when 10000 nodes were added to the tree for the informed RRT* with the optimal local planner. Figure 4.2 shows that it is possible to make use of the optimal local planner to find a trajectory between a starting and goal location, while avoiding obstacles.

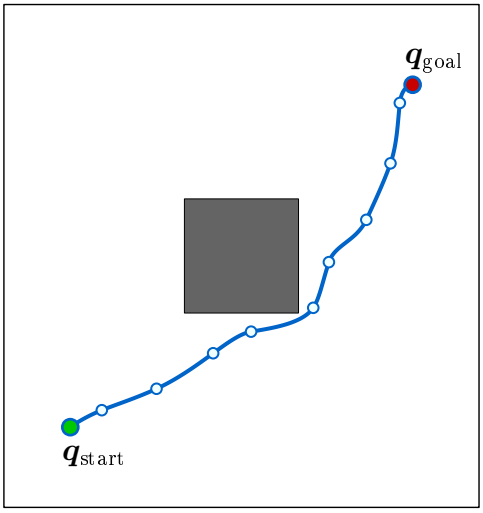


Figure 4.2: The path generated by the informed RRT* with optimal control as local planner, for 10000 nodes.

The informed RRT* was then executed and the time it takes to add each node is logged up until 5000 nodes are added to the tree. This experiment is done for the optimal local planner as well as for straight-line connections between nodes. Straight-line connections can be seen as the most basic form of local planning and will most likely be the fastest way to connect nodes therefore it was used as a benchmark when choosing a local planning method. Figure 4.3 shows the difference in execution time for the motion planning algorithm between the optimal local planner and straight-line connections when more nodes are added to the tree for the same setup shown in Figure 4.2.

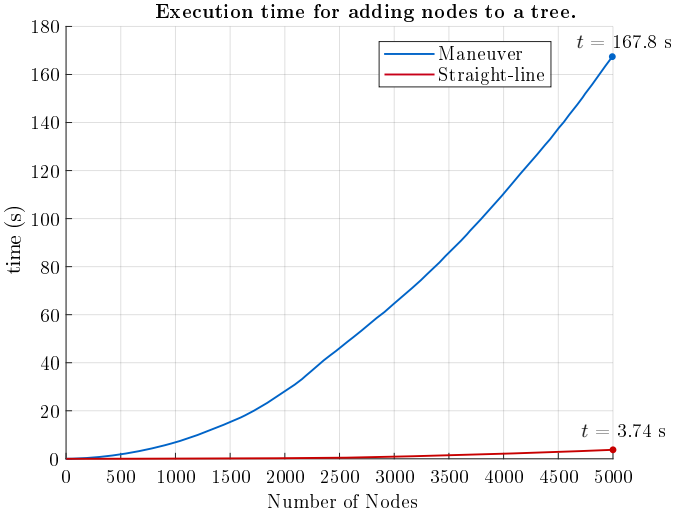


Figure 4.3: The accumulated time of adding more nodes to the tree for the informed RRT* between the optimal local planner and the straight line connections up until 5000 nodes are added.

Figure 4.3 shows that the informed RRT* took 40 times longer to add 5000 nodes to the tree when using the optimal local planner compared to using straight line connections. This is quite significant seeing as the robot model used in this example is very basic. Therefore, when taking the quad-rotor model into consideration, it is clear that the optimal planning method will be extremely expensive in terms of computation time. It was decided to rather look at geometric manoeuvre-based local planning methods, which do not yield as optimal trajectories and does not guarantee that the vehicle will be able to execute these trajectories, but they are a lot less computationally expensive. The manoeuvre-based local planning method used will be discussed in the following section followed by how the quad-rotor will execute trajectories.

4.2 Manoeuvre-Based Local Planner

As discussed in the previous section, an efficient method is needed to connect nodes that also conforms to the dynamic constraints of the quad-rotor. The previous section concluded that using the model of a vehicle and trying to connect states optimally is very inefficient. Therefore, less computationally expensive approach is needed. This section proposes a set of motion primitives or manoeuvres that the local planner can use to connect nodes together and form a trajectory.

A common approach to construct a trajectory from a set of manoeuvres for an aircraft, is to separate the horizontal and vertical motion so that independent horizontal and vertical manoeuvre sets can be designed. This means that it possible to design a set of horizontal manoeuvres and afterwards use the length of the horizontal trajectory to find the corresponding vertical manoeuvres. Section 4.2.1 will discuss how a horizontal trajectory can be constructed by using a set of geometrical primitives that are combined into sequences called Dubins curves [20]. Dubins curves make the assumption that the vehicle moves at a constant velocity and can only turn at a constant radius, to construct a set of six trajectories that can be used to connect nodes together. Section 4.2.2 discusses how the vertical component of the vehicle trajectory can be constructed by assuming the quad-rotor moves at a constant vertical acceleration until it reaches a certain vertical velocity, where it stops accelerating and keeps its velocity constant until it reaches the next node.

Both the horizontal and vertical manoeuvres are not great approximations of the natural manoeuvrability of a quad-rotor which makes it difficult to execute these paths. Section 4.3 will therefore discuss a way to improve the controllers derived in Chapter 2 so that the quad-rotor will be able to execute any given manoeuvre set with acceptable accuracy.

4.2.1 Horizontal Manoeuvres

The horizontal manoeuvres used in this project are based on the manoeuvre set of a so-called Dubins car, introduced by Dubins in 1957 [20]. A Dubins car is the name given to any vehicle that can only move forward in a two-dimensional plane with a constant velocity. Finding the shortest trajectory for Dubins cars is a well-studied problem and has been shown to work well with many different types of vehicles. What makes the Dubins car so attractive is that the shortest trajectory between two states can be determined exactly using relatively simple geometry, which is a lot less expensive in terms of computational time when compared with the optimal trajectory planning of Section 4.1.

A Dubins car can execute one of three commands: “turn left at maximum” (L), "turn right at maximum" (R) and "go straight" (S). A trajectory between any two nodes can be constructed if each node has a fixed orientation by using a combination of these three commands. If a node is given an orientation, it is then possible to connect multiple Dubins trajectories together so that it form a continues trajectory between multiple nodes. Dubins proved that there are only six combinations of these controls necessary that will describes all the shortest trajectories between any two nodes for a Dubins car. These combinations are: RSR, LSL, RSL, LSR, RLR and LRL, which can be seen in Figure 4.4.

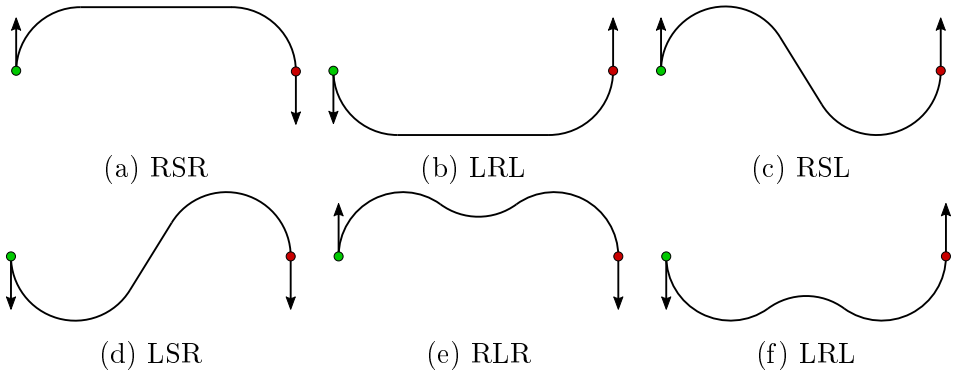


Figure 4.4: The set of all six Dubins trajectories with green as the starting node and red as the goal.

To connect two nodes with a Dubins trajectory, the local planner will consider all six possible sequences and use the one with the lowest cost, where the cost in this case represents the shortest distance between the nodes. When the Dubins manoeuvres are used as local planner for the informed RRT* and 10000 samples are added to the environment, it results in the trajectory shown in Figure 4.5.

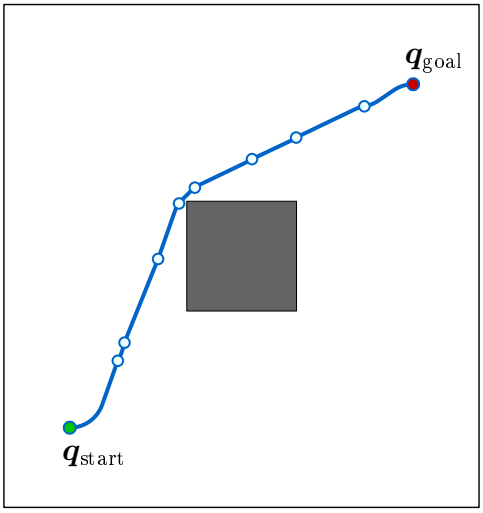


Figure 4.5: The trajectory generated by the informed RRT* with Dubins manoeuvres as local planner, for 10000 nodes.

The same timing test as for the optimal local planner in the previous section was performed for Dubins manoeuvres and the result is shown in Figure 4.6.

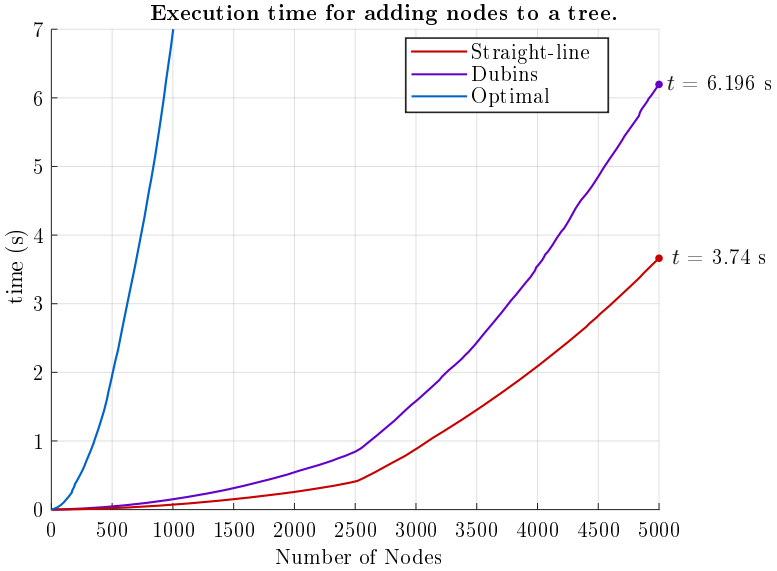


Figure 4.6: The accumulated time of adding more nodes to the tree for the informed RRT* between the optimal local planner, manoeuvre-based local planner and straight-line connections up until 5000 nodes are added.

Figure 4.6 shows that the informed RRT* took two times longer to add 5000 nodes to the tree, when Dubins manoeuvres are used as a local planning method, than straight line connection, which is still ten times faster than with the optimal local planner. It should be noted that the sharp gradient

change at around 2500 nodes in Figure 4.6, for the informed RRT* where both the Dubins manoeuvres and straight-line connections where used, seems to always occur at that same position (at 2500 nodes), even for different testing environments and uniform sampling seeds. It can therefore only be attributed to the way C++ handles memory when the number of nodes saved exceeds a certain amount. Due to its low impact on the results, it was not investigated further.

To have a local planner that fully conforms to the constraints of a quad-rotor that can move in three-dimensional space, it is also necessary to find suitable manoeuvres for the quad-rotor's vertical movement.

4.2.2 Vertical Manoeuvres

It was decided to make use of a parabola-to-line connection to act as vertical manoeuvres for the quad-rotor. To construct these manoeuvres, the quad-rotor was limited to move in a fixed way between any two states. This movement can be summed up as follows:

1. The quad-rotor should not exceed a maximum vertical acceleration;
2. It should accelerate or decelerate at a constant rate from its initial position until it reaches a certain velocity;
3. after which it will keep this constant velocity until it reaches its goal location.

By limiting the quad-rotor's movement in this way, it makes it possible to construct vertical manoeuvres. Figure 4.7 shows an example of a parabola-line trajectory that conforms to the limitations. (L_1, z_1) represents the starting node and (L_2, z_2) is the goal node. L represents the total horizontal distance travelled and z refers to the vertical position of the quad-rotor. To construct one of these vertical trajectories, it is necessary to find the point, (L_x, z_x), where the parabola intersects the straight line, or where the quad-rotor stops accelerating.

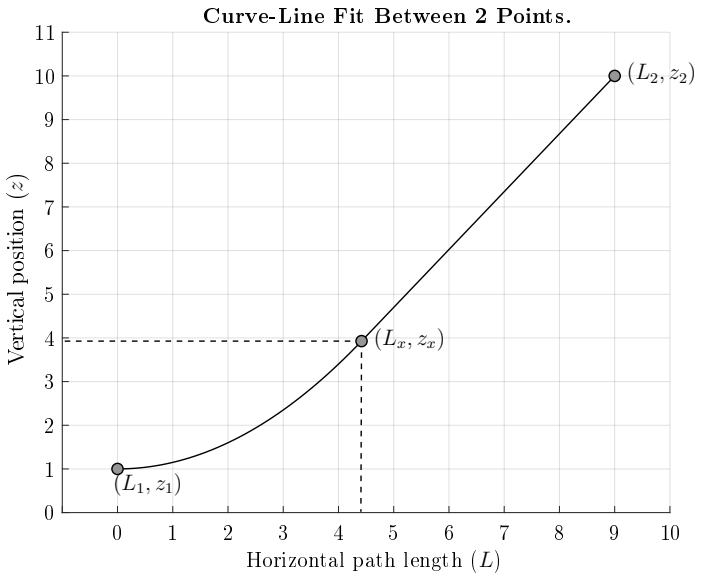


Figure 4.7: Fitting a parabola to a line between two given points.

There are two different ways to construct such a trajectory, depending on the information known about the start and goal nodes. If the vertical velocity at the start is known, but not the goal velocity, (this happens only when a new node is sampled) the quad-rotor will then accelerate at the given maximum, A_z, until it reaches z_x where it will move with a constant vertical velocity until it reaches z_2. This velocity will then be added to the goal state. L_x, from the intersection point between the parabola and straight line, can be calculated as

$$L_x = L_2 \pm \frac{\sqrt{A_z({L_2}^2 A_z + 2{V_L}^2(z_1 - z_2) + 2L_2 V_L V_{z_1})}}{A_z}, \quad 0 \leq L_x \leq L_2, \quad (4.18)$$

where V_L is the constant horizontal speed of the quad-rotor when executing a Dubins trajectory and V_{z1} is the vertical velocity at the start. z_x is then obtained by substituting L_x into the equation for a parabola, which gives

$$z_x = \frac{L_x(2V_L V_{z1} + L_x A_z)}{2V_L^2}. \quad (4.19)$$

The velocity at the goal state can then be calculated as

$$V_{z2} = \frac{z_2 - z_x}{L_2 - L_x} V_L. \quad (4.20)$$

The derivation of these equations is presented in Appendix B. To calculate the intersection point for the case where the vertical velocity at the goal is given, it is assumed that the quad-rotor can accelerate at any speed up till A_z. To

find L_x and z_x, it is necessary to calculate the constant acceleration a_z that will result in the desired goal velocity. This is calculated as

$$a_z = \frac{V_L(V_{z1} - V_{z2})^2}{2L_2V_{z2} + 2V_Lz_1 - 2V_Lz_2}, \qquad 0 \leq a_z \leq A_z, \tag{4.21}$$

where V_{z2} is the vertical velocity at the goal state. The intersection point, (L_x, z_x) can then be obtained by

$$L_x = \frac{V_L(V_{z2} - V_{z1})}{az}, \qquad L_1 \leq L_x \leq L_2, \tag{4.22}$$

and

$$z_x = \frac{V_Lz_2 - L_2V_{z2}}{V_L} + \frac{V_{z2}(V_{z2} - V_{z1})}{az}. \tag{4.23}$$

With the intersection point obtained, it is now possible to calculate the total trajectory length between any two nodes. The trajectory length can then be used during cost calculations, which is necessary to add and rewire nodes for the sampling-based motion planning algorithms of Chapter 3. The length of the parabola section is given by

$$P_{\text{dist}} = \int_{L_1}^{L_x} \sqrt{1 + f'(L)^2}, \tag{4.24}$$

where $f'(L)^2$ is the derivative of the parabola equation, $z = aL^2 + bL + c$. The closed form solution of the integral is given by

$$P_{\text{dist}} = \frac{(2aL_x + b + 1)^{\frac{3}{2}} - (b + 1)^{\frac{3}{2}}}{3a}. \tag{4.25}$$

The length of the straight-line section can then be calculated with

$$S_{\text{dist}} = \sqrt{(z_2 - z_x)^2 + (L_2 - L_x)^2}. \tag{4.26}$$

By using Equation 4.25 and 4.26, the total length of a trajectory between two nodes is obtained.

Adding the vertical manoeuvres to the local planner and executing the same timing experiment as before resulted in a small increase in execution time compared to just the Dubins manoeuvres, which can be seen in Figure 4.8. The planner with both horizontal and vertical manoeuvre together is also much faster than the one with the optimal local planner.

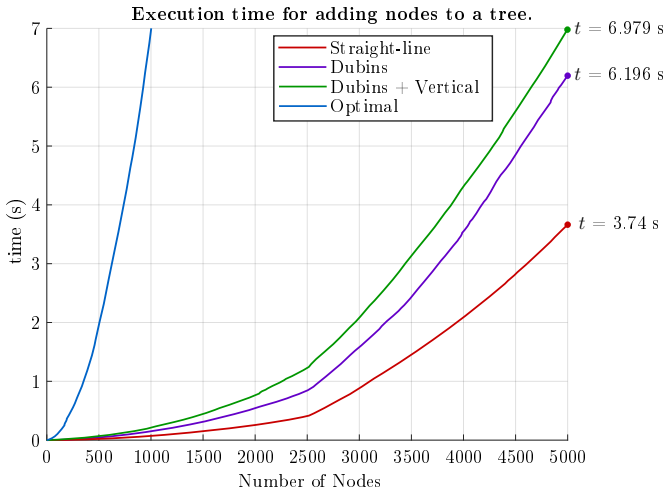


Figure 4.8: The accumulated time of adding more nodes to the tree for the informed RRT* between the optimal local planner, manoeuvre-based (horizontal and vertical) local planner and straight-line connections up until 5000 nodes are added.

The manoeuvres are implemented as local planner for the informed RRT* alongside the horizontal Dubins manoeuvres. The vertical acceleration (A_z) was limited to 0.3 m·s^{-2} and the planner was executed for 10000 nodes in three-dimensional space. Figure 4.9 shows the vertical trajectory between nodes with respect to the total length of the horizontal Dubins trajectory and Figure 4.10 shows the full trajectory found in three-dimensional space.

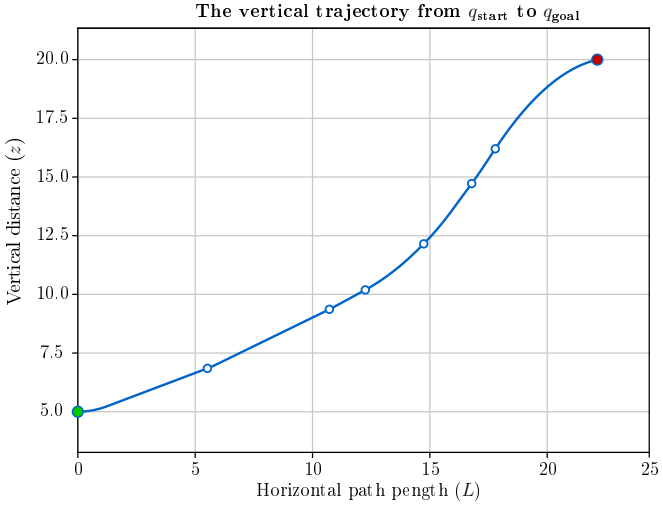


Figure 4.9: The vertical trajectory between a start (green) and goal (red) node with the horizontal distance.

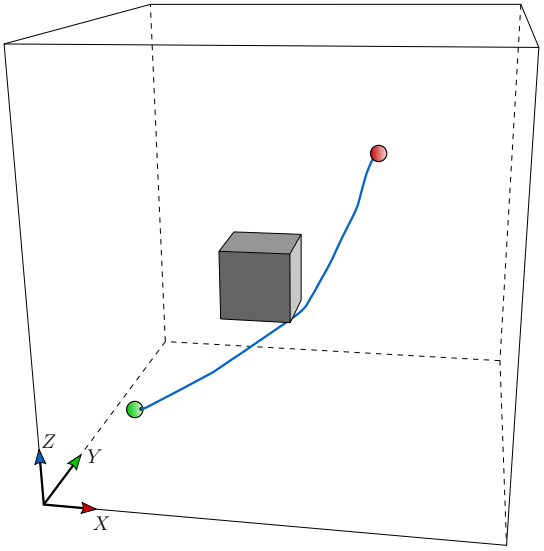


Figure 4.10: The 3-D trajectory generated by the informed RRT* when 10000 nodes are added to the tree for vertical and horizontal manoeuvres.

By using both the vertical and horizontal manoeuvre sets as local planning method for the motion planning algorithms of Chapter 3, trajectories are generated that should conform to the constraints of the quad-rotor. To determine if the quad-rotor are able to accurately execute the trajectories generated by motion planning algorithm, these trajectories are given as input to the aircraft model and controllers from Chapter 2. The following section will discuss the results found and implement feed-forward control to improve trajectory tracking.

4.3 Feed-Forward Control

Consider Figure 4.11, which shows the planned trajectory tracking of the quad-rotor in the inertial axis system. The trajectory was generated by the informed RRT* with the manoeuvres-based local planner. The quad-rotor then tries to follow the trajectory by using the pre-existing controllers mentioned in Chapter 2 as feedback control. The position coordinates of the full trajectory, at each time step, is given to the system as inputs. It is assumed that the quad-rotor moves at a constant horizontal velocity of 1 $\text{m}\cdot s^{-1}$ and has a maximum vertical acceleration of 0.3 $\text{m}\cdot s^{-2}$.

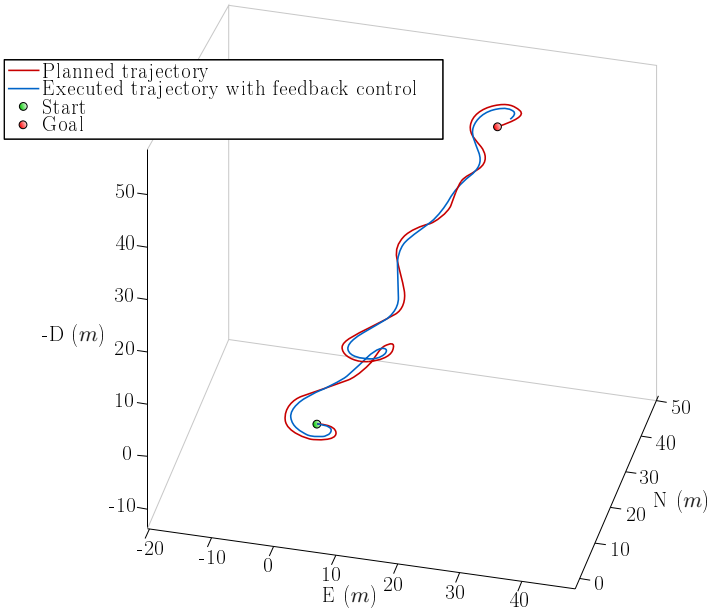
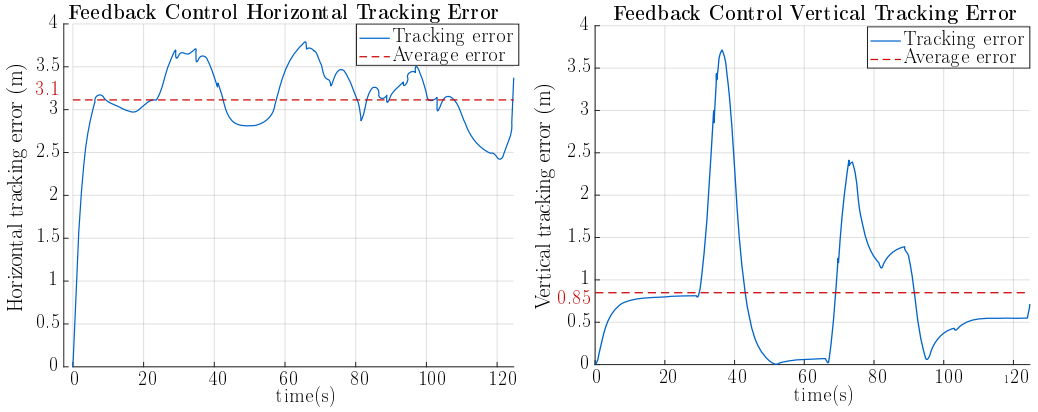


Figure 4.11: The trajectory tracking of the non-linear quad-rotor model when feedback control is used.

Figure 4.11 shows the planned trajectory (red) and the measured position output (blue) of the non-linear quad-rotor model. It is clear that the feedback controllers are able to follow the given trajectory, but with a clear offset in position. Due to the way the controllers and manoeuvres are designed, the system should be able to independently follow the horizontal and vertical coordinates of the input trajectory. Therefore, the position error for both the horizontal and vertical trajectories are calculated separately. Figure 4.12 shows the position error of the feedback control system for the non-linear quad-rotor model, at each time step, for the trajectories shown in Figure 4.11.



(a) Horizontal position tracking error. (b) Vertical position tracking error.

Figure 4.12: The position tracking error of a given trajectory for the feedback control system of the non-linear quad-rotor model.

Figure 4.12 indicates that the feedback control system on average gives a position tracking error of 3.1 m for horizontal motion with a peak around 3.7m when the quad-rotor has to make a constant turn and only drops below 3 m when it moves in a straight line. The tracking error for vertical motion has an average error of 0.85 m. The error increases when the quad-rotor has to quickly change its vertical position and reached a peak of 3.7 m, but has almost zero error when there is little vertical displacement.

It is clear when looking at both Figures 4.11 and 4.12 that by only using the feedback controllers originally designed for the quad-rotor, it was not able follow the given trajectory with acceptable accuracy. This means that when the motion planner find a possible solution trajectory, it can not be certain that this trajectory will be collision free because of the large deviation from the given trajectory. When using the feedback controllers, the quad-rotor also fails to reach the goal at the given time (seen in Figure 4.11), which is a big problem for planning in dynamic environments, where the time to reach a position is important. From these issues, it is clear that the pre-existing control system needs to be improved. This improvement was done by adding velocity inputs to the model as feed-forward control.

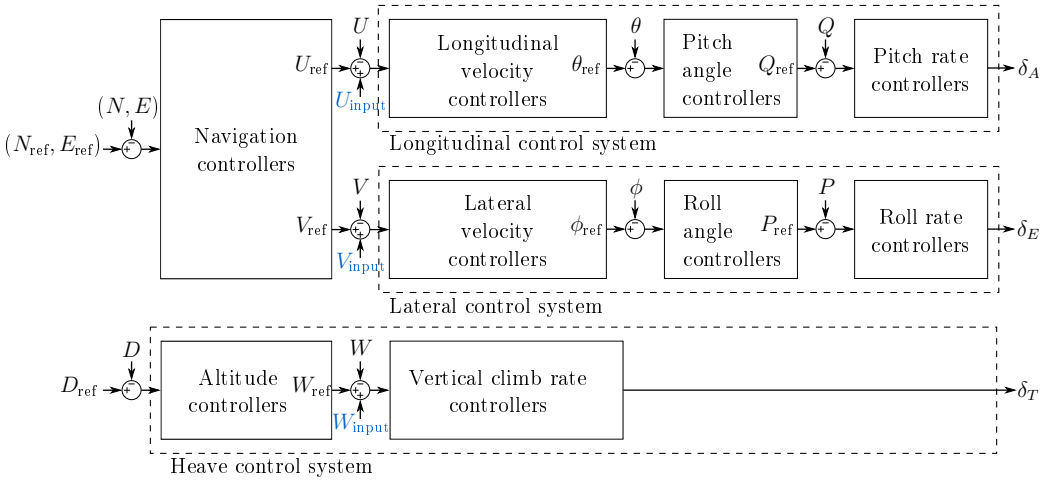


Figure 4.13: Block Diagram of the pre-Existing flight control system with feed-froward inputs (blue) added to the system.

Feed-forward control makes use of the the knowledge of what the velocities should be to execute a trajectory and feed it to the inner loops, velocity commands. If the quad-rotor is able to follow these inputs perfectly, then there will be no errors in the outer loops. If the inner loops do not follow their feed-forward references perfectly, or there are disturbances, the outer loops will try to correct these errors. Figure 4.13 shows where feed-forward control fits into the pre-existing controllers from Chapter 2. U_{input}, V_{input}, and W_{input} are the feed-forward velocity inputs in the inertial axis frame needed, at each time

step, that represents any given trajectory. These inputs are given directly to the velocity controllers of the system. Due to the use of geometric manoeuvres chosen, it is possible to calculate the velocity components at any point on the trajectory. When the velocity inputs are given to the model, as well as the position inputs for feedback control, the quad-rotor is able to follow the trajectory with a significant increase in accuracy. Figure 4.14 shows the trajectory tracking of the quad-rotor when feed-forward control is added to the system, for the same planned trajectory as before.

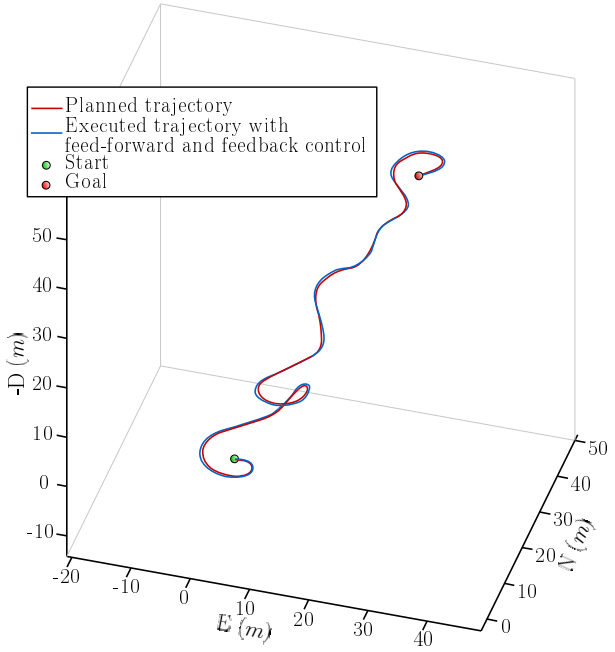
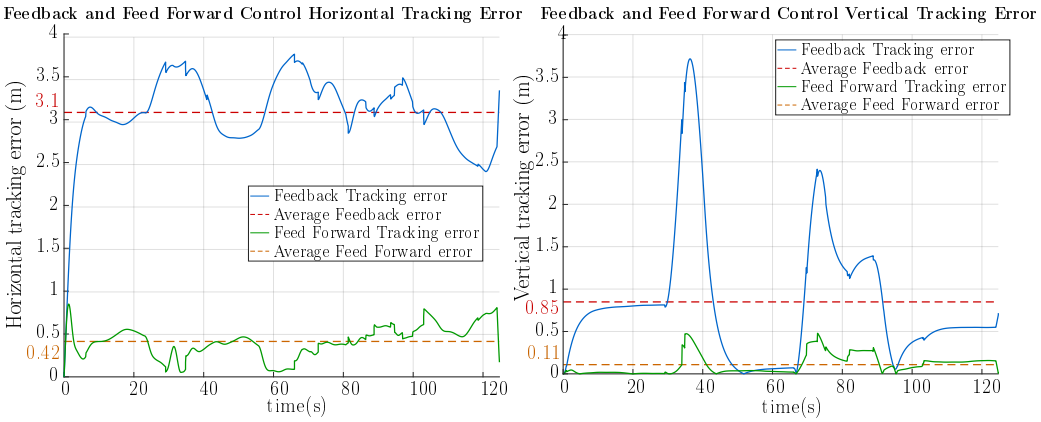


Figure 4.14: The trajectory tracking of the non-linear quad-rotor model when both feedback control and feed-forward control are used.

The position error at each time-step was calculated the same way as before and the result is shown in Figure 4.15. Figure 4.15 shows that by adding the feed-forward control to the same system results in much better trajectory tracking. The average horizontal position error dropped to 0.42 m with a peak deviation of 0.86 m, which is a significant improvement when compared to only using feedback control. The average vertical position error is calculated as 0.11 m with a peak of 0.47 m. If it is necessary to further decrease the tracking error, the maximum turning radius of the Dubins manoeuvres can be increased as well as the vertical acceleration of the vertical manoeuvres can be decreased, but considering the size of the quad-rotor which has a width of 1.5m, an average error of 0.86 m is seen as acceptable.



(a) Horizontal tracking error.

(b) Vertical tracking error.

Figure 4.15: The position tracking error of a given trajectory when feed-forward control is added to the system.

Now that the manoeuvre-based local planning method is explained and the position tracking is verified, it can be implemented as local planner for the RT-RRT* motion planning algorithm from Chapter 3. The next chapter will run simulations to determine how well this motion planning solution will work in dynamic environments and if it is possible to be used as planner for the chosen testing vehicle.

Chapter 5

Implementation and Testing

This chapter introduces the full motion planning solution for a quad-rotor in dynamic environments by combing the RT-RRT* motion planning algorithm and the manoeuvre-based local planning method, and testing the functionality in different environmental conditions. Before these simulations are presented, it is first necessary to discuss the hardware and simulation environment used as well as any changes made to the standard algorithms. Only after these aspects are discussed is it possible to make accurate conclusions on the ability of the planning algorithm.

The final implementation is tested in three-dimensional space for different environments and timing data is then gathered to see whether this planning algorithm is suitable for on-board implementation and which aspects should be improved. This chapter then goes on to introduce obstacle estimation and future prediction for dynamic obstacles, which is then implemented along side the motion planner to determine if it will improve obstacle avoidance as well as how the planner will cope with uncertainty around obstacles.

The chapter concludes with a brief discussion of the execution time of the algorithm and possible aspects for improvement.

5.1 Simulation Environment

All the examples of previous chapters, which includes the aircraft model, motion planning algorithms and local planning method, made use of the following hardware and software to generate the results. Some effort went into using efficient programming languages and plotting packages so that the results could be measured more effectively, but little effort went into optimising the code written for this project to increase performance.

5.1.1 Hardware

The hardware used to run the simulations is a basic desktop computer with the following specifications:

- Operating System: 64x Microsoft Windows 10 Pro
- Processor: Intel Core i5-6500 CPU @ 3.2 Ghz, 4 Cores
- Memory: 8.00 GB DDR3L 1600
- GPU: AMD Radeon R7 350X
- Storage: 125 GB SSD

5.1.2 Software

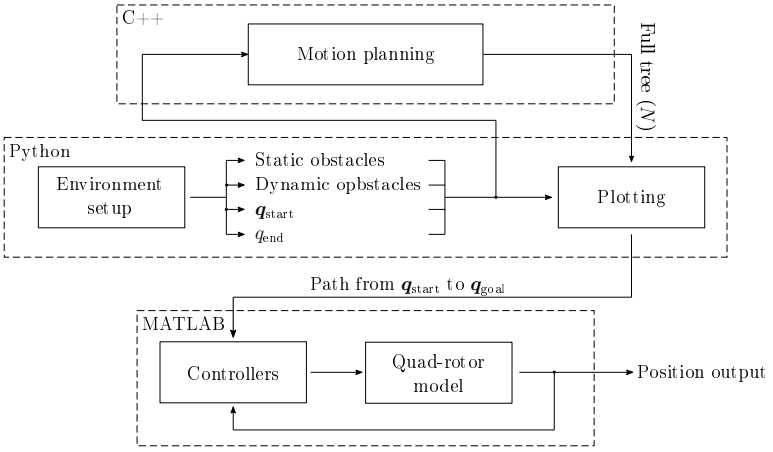


Figure 5.1: The software simulation setup.

The software used for simulation is a combination of Python, C++ or MATLAB, depending on the type of problem that needed to be solved. Python and C++ is used together for all the motion and local planning computations through the use of a Python extension called CYTHON, which allows the calling of C/C++ functions and variables from Python (for more information refer to [21]). This means that all complex operations could be done more efficiently through C++ and all plotting could be easily done through Python packages. The three-dimensional and animated plotting was done with a Python package called Mayavi [22], due to its ability to render three-dimensional structures quickly and has the potential to animate plots, which was used to simulate real-time operations. This is necessary to visualise dynamic obstacles and to see the RT-RRT* in action. MATLAB/Simulink is used to implement the quad-rotor model as well as testing to see if the vehicle controllers were able to track the trajectories generated by the Python/C++ code. Figure 5.1 shows a diagram

of the full simulation software setup and how each environment interacts with one another.

With the software setup shown in Figure 5.1, it was possible to visualise, test and verify the workings of the RT-RRT*, local planner and quad-rotor controllers in any given environment.

5.1.3 Visualisation

Figure 5.2 shows an example of the visualisation environment where a simulated quad-rotor moves and avoids obstacles by following a planned trajectory.

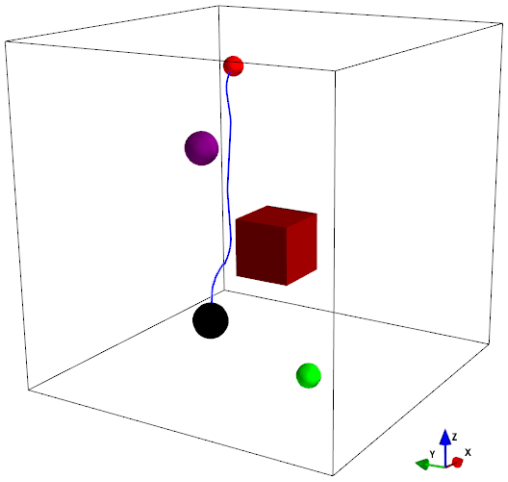


Figure 5.2: The visualisation of an example simulated environment (see text for details).

The green sphere in Figure 5.2 represents the starting position of the quad-rotor and the red one, the goal. The black sphere is the moving agent or quad-rotor and the blue line is the current trajectory being executed. The red cube is a static obstacle and the purple sphere is a dynamic obstacle moving with a predefined velocity. The XYZ axis represents the east (X), north (Y) and down ($-Z$) axis of the inertial frame.

The simulated setup presented here will be used by all the experiments in the latter part of this chapter, to make accurate conclusions on the performance of the proposed planning solution. The next section will present the final motion implementation, including the changes made to the algorithms described in the previous chapters.

5.2 Implementation

This section will introduce the final planning solution, that was implemented and tested. It starts off by describing the change made to the RT-RRT*, to reduce the time to find high quality, low cost trajectories. This was done by changing the way the RT-RRT* handles rewiring. This section then presents the parameter values for the RT-RRT*, that were used during simulation.

The section then moves on to explaining a change made to the horizontal Dubins trajectories of Chapter 4 that will increase the convergence rate of the solution trajectory, before presenting the parameter values for the local planner that were used in the simulation.

5.2.1 RT-RRT*

For the version of the RT-RRT* implemented in this project, we made changes to the way nodes are added to the tree and how the rewiring of Algorithm 2 is performed. The rewiring version we implemented can be referred to as a colour-based rewiring strategy and allows the RT-RRT* to rewire nodes that are in collision with dynamic obstacles so that when they become available, they reduce the time it takes the planner to find low-cost trajectories, as opposed to the method mentioned in Chapter 3.

5.2.1.1 Colour-Based Rewiring Strategy

The idea behind the new rewiring strategy is to reduce the time it takes the planner to find a new high-quality solution when the current best solution becomes blocked by a dynamic obstacle, which happens when the current measured position of a dynamic obstacle in time causes collision with the solution trajectory. The way the RT-RRT* of Chapter 3 handles dynamic obstacles, is by assigning a cost of infinity to nodes that are blocked by dynamic obstacles. This means that when a node in the middle of the tree becomes blocked, it, and all the nodes attached to it, will now have a cost of infinity. The problem with this approach is that large portions of the tree, which now has a cost of infinity, becomes unusable in finding a collision free trajectory from the start node. These portions only becomes available when their nodes are rewired such that there is a collision-free connection to the start node. The planning algorithm then has to use the available tree to find a new solution, which can result in low-quality trajectories.

It was therefore decided to develop a new rewiring strategy to mitigate this problem. The designed solution, can be described as a colour-based rewiring strategy. The difference with our method is that when a new node is added, or an old one is rewired, it does not assign a cost of infinity to the node if it is blocked, which happens when the node, or the trajectory from the start node

to this node, overlaps with an obstacle. It rather assign the cost normally and make use of a priority-based system to assign a parent to a node. This means that a node, that is not blocked will always be seen as a lower-cost parent, compared to any nodes that is blocked. This will result in low-cost trajectories around an obstacle, but also allow the nodes inside the obstacle region to have a low cost as well. When rewiring nodes with this method, and an obstacle moves to a new position, the nodes that were previously inside the obstacle region open up and the algorithm can now make use of these nodes to update the current best solution and in this case, as opposed to the previous method, these nodes has already been assigned the lowest cost parent.

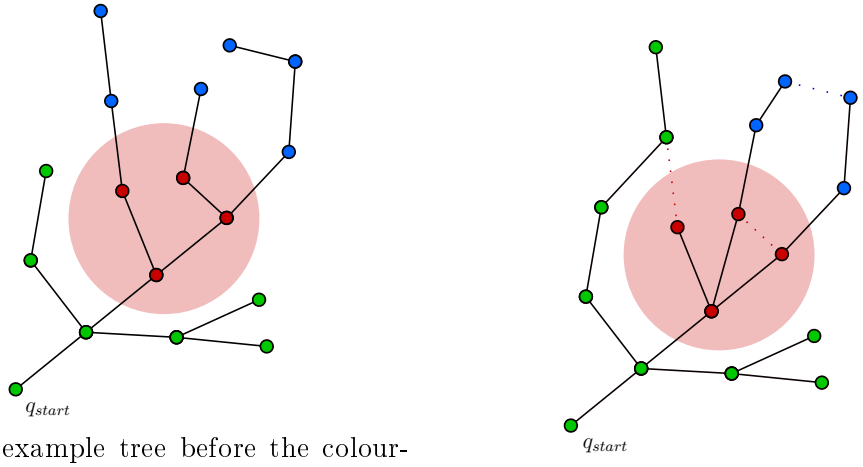
This method works by assigning a colour to all the neighbours of Algorithm 2 before searching for the least-cost parent. The colour of a node depends on their relationship with the dynamic obstacles in the environment. Algorithm 6 shows how each colour is assigned. If a node, $\boldsymbol{q}_i$, lies inside the collision area of an dynamic obstacle, it is seen as a "Red" node. If any of the nodes in the trajectory that runs from $\boldsymbol{q}_{\text{start}}$ to $\boldsymbol{q}_i$ lie in the collision area, but not $\boldsymbol{q}_i$, then $\boldsymbol{q}_i$ is given the colour "Blue" and if the trajectory to reach $\boldsymbol{q}_i$ does not collide with any dynamic obstacles, it is seen as "Green".

Algorithm 6: Assign a colour to node for rewiring

Input : $\boldsymbol{q}_i$
1 **if** $\boldsymbol{q}_i$ *is inside* X_{obs} **then**
2 | $\boldsymbol{q}_i \leftarrow \boldsymbol{q}_i^{\text{Red}}$
3 **else if** *any parent of* $\boldsymbol{q}_i$ *is inside* X_{obs} **then**
4 | $\boldsymbol{q}_i \leftarrow \boldsymbol{q}_i^{\text{Blue}}$
5 **else**
6 | $\boldsymbol{q}_i \leftarrow \boldsymbol{q}_i^{\text{Green}}$

This colour-based system is used in both adding nodes to the tree (Algorithm 2, line 7) as well as during rewiring (Algorithm 2, line 9 and 10). Consider the RRT* of Section 3.4.1. When a new node is sampled, it tries to connect it to the node in the tree with the lowest cost. The new strategy, on the other hand, uses the colour-based system to add nodes to the tree. If the newly sampled node, $\boldsymbol{q}_{\text{new}}$, lies inside the collision region of a dynamic obstacle, it does the same as the RRT* and finds the least-costly parent to connect to. If $\boldsymbol{q}_{\text{new}}$ lies anywhere else in the environment, it first tries to find the least-costly green neighbour to connect to. If no such neighbour exists, it then tries to find a blue neighbour. If no suitable neighbour is found, it will finally look for a red neighbour to connect to. The same is done for the rewiring operation. It should be noted that no red trajectories will be considered as a final solution between the start and goal nodes. The colour-based rewiring solution is illustrated in Figure 5.3. Figure 5.3a shows a tree before rewiring and Figure 5.3b the tree

after rewiring has been performed.



(a) An example tree before the colour-based rewiring strategy is performed, where each node has been assigned a colour. The red circle is seen as a collision area.

(b) The tree after the rewiring strategy has been performed, where the dotted lines show the previous connections.

Figure 5.3: The colour-based rewiring solution.

Figure 5.4 shows a fully-rewired tree with 2000 nodes when the colour-based rewiring strategy is used for a two-dimensional environment with straight-line connections as local planner. The environment consists of one dynamic obstacle that is unable to move and the motion planning algorithm should find a trajectory around it. From Figure 5.4 it is clear that the new rewiring strategy is able to find a collision-free trajectory (blue) around the obstacle and that all nodes inside the collision area are rewired but does not extend outside of the collision area.

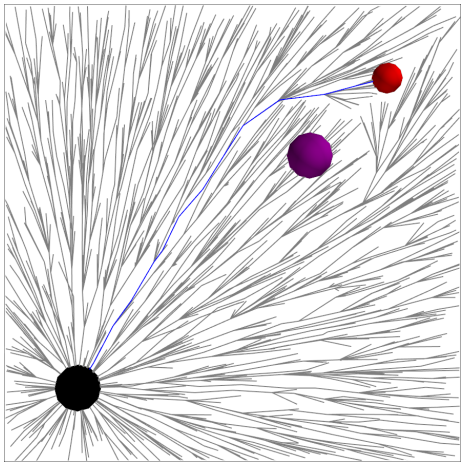


Figure 5.4: Colour-based rewiring performed on 2000 nodes in the tree.

By doing the expansion and rewiring in this way, all nodes in the tree will be connected to its most optimal parent node, based on its colour. When a dynamic obstacle moves to a new position, the algorithm wastes no time to find a new solution that is already optimised, which should decrease the final distance travelled by the agent to reach the goal location and will be verified in Section 5.3.

5.2.1.2 Parameters

The following parameters that are necessary to run the RT-RRT* were found to work reliably for most environments:

- $r_n = 8$ m in Algorithm 2 (the maximum neighbour distance).
- $\beta = 10$ in Equation 3.1 (the amount of times samples are drawn from the ellipsoidal subset).
- $\alpha = 0.01$ in Equation 3.1 (the amount of times the goal region is sampled).
- $\tau = 0.03$ s in Algorithm 1 (the time used for expansion and rewiring).

Before any simulation can be performed, it is necessary to explain the changes made to the local planning method as well as the parameter values used.

5.2.2 Local Planning Method

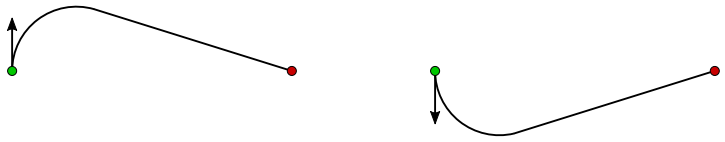
The Dubins curves used for horizontal trajectories of Chapter 4 were constructed by fitting circle segments and straight lines together to from the Dubins manoeuvres as explained in Section 4.2.1. Refer back to Figure 4.4 for a visual reminder of the Dubins trajectories. To construct a Dubins trajectory between two nodes, it is necessary to know the orientation of the nodes so that the local planner knows where to start and end the circle segments. To assign an orientation to a node, it should be be included in the motion planner's random sampling step, so that when the local planner has to find a Dubins manoeuvre to connect two nodes, it can use the randomly-sampled orientation to know when to start and stop the left and right turns. The problem with adding another dimension to the sample space is that it increases the number of nodes needed in the tree to find high-quality trajectories.

It was decided that, in order to effectively avoid dynamic obstacles and quickly reduce the cost of the final trajectory, the quad-rotor does not need to reach its goal with a specific orientation. This removes the need for sampling the orientation and decreases the time it takes the planner to find a high-quality

trajectory. This was done by changing the Dubins manoeuvres slightly when a new node needs to be added to the tree. The orientation of a newly-added node is calculated based on the connection between this node and its lowest-cost parent, instead of sampling the orientation, before fitting a trajectory between the two nodes.

5.2.2.1 Dubins Trajectory Changes

To change the Dubins curves to work in this way, the final turn sequence for all the six Dubins trajectories is removed. In other words, any two nodes will now be connected by an initial curve (left or right turn) followed by a straight line, but no final curve. By removing the final sequence from the Dubins trajectory, the shortest trajectory between any two nodes, can now be described with only two trajectories, shown in Figure 5.5. After the least-costly sequence has been found, the new node is added to the tree and the orientation to reach this node is calculated, so that the local planner can use the normal Dubins trajectories when rewiring is performed.



(a) RS (b) LS

Figure 5.5: The Dubins trajectories without a fixed goal orientation.

By changing the Dubins curves in this manner, it is not necessary to sample the orientation of a node, which should, reduce the time the motion planner takes to find a high-quality solution and will be verified in Section 5.3.2.

5.2.2.2 Parameters

The following parameters values are chosen for the horizontal Dubins trajectories:

- $r_d = 4$ m (maximum turning radius)
- $V_L = 1$ m·s^{-1} (constant horizontal velocity)

For the parabola-line vertical manoeuvres the following parameter values where chosen:

- $A_z = 0.3$ m·s^{-2} (constant acceleration for parabola segment)
- -5 m·s$^{-1} \leq V_z \leq 5$ m·s^{-1} (vertical velocity for line segment)

With the parameter values chosen as well as the changes made to the RT-RRT* and local planner, the results of the full motion planning solution for a quad-rotor in dynamic environments can now be evaluated.

5.3 Simulations

To validate that the RT-RRT* motion planning algorithm and the manoeuvre-based local planner, is a viable planning solution for a quad-rotor in dynamic environments, some simulations that represent real-world problems, need to be performed. This section will start off by comparing the new colour-based rewiring strategy to the one proposed by Naderi, Kourosh and Rajam for the RT-RRT* [18], after which it will compare the updated Dubins trajectories to the original method to determine if the changes are successful. This section will then evaluate a number of example environments of rea- world problems to test the effectiveness of the solution. Obstacle estimation and future prediction is then added to the planning algorithm so see if it will improve obstacle avoidance. This section end off by comparing the computational time of some main operation of the motion planner, to see where possible improvements can be made.

5.3.1 Rewiring Strategy

The rewiring strategy, as mentioned previously, was designed to reduce the time it takes the planning algorithm to find a higher quality alternative trajectory when an obstacle blocks the current solution, which reduces the total distance travelled by the quad-rotor to reach the goal. Therefore, to test the effectiveness of the new strategy, the average cost to reach a given goal location for both strategies is measured for multiple environment setups, each one with an increasing number of dynamic obstacles that needs to be avoided. We expect the new strategy to perform the same as the base solution for environments with zero dynamic obstacles and become increasingly better when more obstacles are added to the environment.

To determine if the new strategy works as expected, the motion planning algorithm was executed in environments consisting of 0 to 5 dynamic obstacles, for both rewiring strategies. For each environment, the simulation was executed

100 times and the trajectory length of the final solution was measured for each rewiring strategy. The mean and standard deviation of the trajectory lengths is calculated for the 100 simulations and the results are shown in Table 5.1.

Table 5.1: A comparison between the original RT-RRT* and the proposed colour-based rewiring strategy for environments with increasingly more dynamic obstacles.

	Original		**Proposed**	
Number of Obstacles	μ (m)	σ (m)	μ (m)	σ (m)
0	27.87	1.32	27.73	1.30
1	28.90	2.49	28.18	1.92
2	29.12	2.68	28.93	2.88
3	30.15	4.30	28.85	2.43
4	30.89	4.81	29.05	2.97
5	31.96	6.55	29.21	2.71

Table 5.1 shows that for 0 obstacles in the environment there are little to no difference in the trajectory lengths for both rewiring strategies, as expected. The difference in trajectory length is much more noticeable with the increase in obstacles. The average trajectory length found for both strategies increases with the addition of more obstacles, with the original rewiring strategy increasingly more. The biggest difference comes in when comparing the standard deviation of both cases. The newly-proposed rewiring strategy has a maximum deviation of 2.97 m from the mean, which is much less than the original strategy that has a maximum deviation of 6.55 m. This means that the new rewiring strategy has a more reliable outcome.

The distribution of the different trajectory lengths found, for each rewiring strategy, is shown in Figure 5.6 for the case where 5 obstacles are present in the environments. From this figure it is clear that the proposed rewiring strategy is able to find trajectories close to its mean and do not deviate far from it, whereas the original strategy, although it has a high probability to find a trajectory with a length close to the mean, it often deviate far from it and therefore is not able to reliably find high-quality solutions. The simulations thus shows that the newly proposed colour-based rewiring strategy is able to out-perform the original by not only finding higher quality trajectories, but also finding such trajectories consistently even with the addition of dynamic obstacle to the environment.

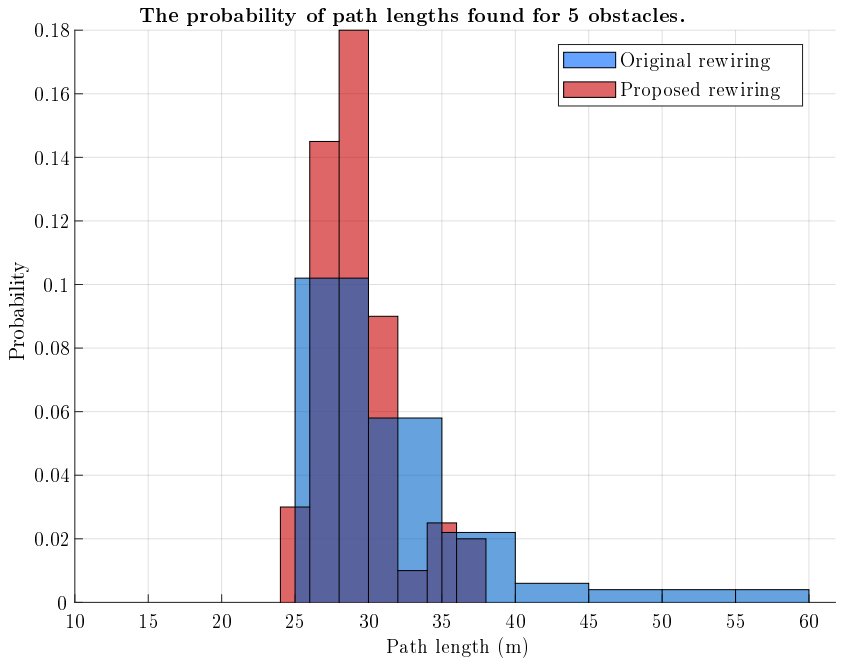


Figure 5.6: The normalised histogram plot of the trajectory lengths found by the two different rewiring strategies when 5 obstacles are present in the environment.

Before the final motion planning solution can be tested, it is first necessary to verify whether the updated Dubins curves are capable of reducing the number of samples necessary to find the same quality trajectory.

5.3.2 Dubins Trajectories

To determine if removing the orientation sampling from the Dubins curves, when adding a new node to the tree, improves the convergence rate of the solution, the RRT* planning algorithm from Section 3.4.1 was executed with the Dubins local planner for the case where the orientation is sampled as well as where it is not sampled, in the same environment. The planning algorithm was allowed to run until the total trajectory length of the final solution was below a given threshold (chosen as 31 m) and the number of nodes in the tree was measured. This simulation was executed 1000 times for each case, and Figure 5.7 shows the results.

Figure 5.7 shows that for the normal Dubins trajectories (blue), where the orientation is sampled, an average tree size of 945 nodes is needed to find a trajectory below a length of 31 m, with a standard deviation of 260 nodes. When not sampling the orientation and using the reduced manoeuvre set (red), it gave an average tree size of 210 nodes with a standard deviation of 97 nodes. This means that the removal of orientation sampling from the Dubins curves

significantly reduces the number of samples needed to find a certain quality trajectory. Having fewer nodes in the tree means that it took less time to find a trajectory with the same quality and proves that by not sampling the orientation, will increase the convergence rate of the final trajectory.

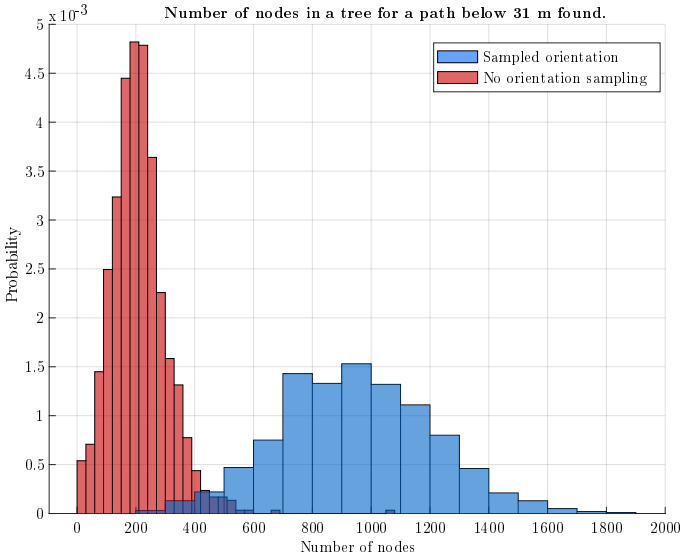


Figure 5.7: The normalised histogram plot of the number of nodes necessary to find a trajectory below 31 m for the Dubins local planner when the orientation is sampled versus when as well when it is not sampled.

With all the necessary additions explained and verified, the full motion planning solution can now be tested.

5.3.3 Full Planning Solution

The aim of this thesis is to develop a motion planning solution for a quadrotor in dynamic environments. To verify that the RT-RRT* alongside the manoeuvre-based local planner is a viable solution, a few simulations will be presented in this section to test the effectiveness of this solution.

The first simulation is a simple environment with one static and one dynamic obstacle. The motion planning solution should try to find a trajectory around the static obstacle as well as be able to rewire the trajectory when the dynamic obstacle moves in the way of the current best trajectory. This test scenario is chosen as baseline to determine if the planner will be able to:

- Find a trajectory to goal that is collision free,
- reduce the cost of the trajectory while the agent is moving

- and rewire when a moving obstacle blocks the current trajectory.

The second simulation consists of multiple dynamic obstacles with random starting positions and velocities, spread out over the environment. The planner is tasked to find a collision free trajectory through the clutter by rewiring the trajectory to the goal when necessary. The purpose of this test is to verify that the motion planning solution can rewire and re-plan quickly enough to move safely through cluttered environment.

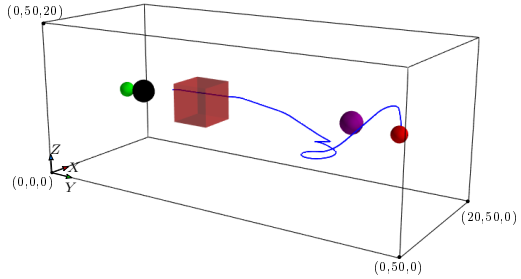
The third simulation will add obstacle estimation and future prediction to the planning method by the use of a Kalman filter that can track the movement of a dynamic obstacle. This is then implemented on the same environment as simulation two to determine if it will improve obstacle avoidance.

Each simulation is executed 100 times so that the distribution of total distance travelled by the agent and the distribution of the number of collisions can be analysed.

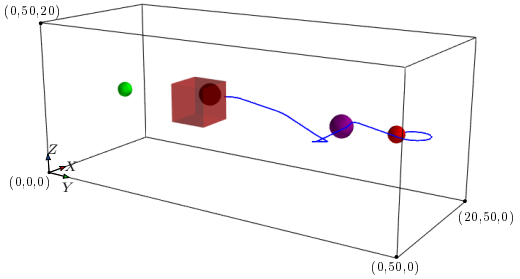
5.3.3.1 Simulation 1: Sparse Environment

Figure 5.8 shows the planned trajectory at different time steps and how it got updated as the agent (quad-rotor) moves through the environment. The environment consists of one static (red cube) and one dynamic obstacle (purple sphere). The static obstacle is directly between the start (10 m, 5 m, 10 m) and goal (10 m, 40 m, 10 m) positions, which forces the agent to move around it and the dynamic obstacle is chosen to eventually move across the planned trajectory. This forces the planning algorithm to rewire the trajectory to goal at some point in time to avoid the dynamic obstacle. This specific scenario is used to test the performance of the planning algorithm by measuring the total distance travelled by the agent to reach the goal as well as if the agent was able to avoid the obstacles.

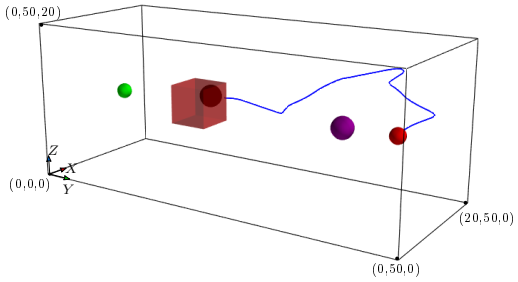
Figure 5.8a shows the first trajectory found after a few sampling and rewiring iterations. The quality of the trajectory then increases form Figure 5.8a to 5.8b as more samples are added to the tree and rewiring is performed, while the agent moved along the initial trajectory. At the point in time for which Figure 5.8b is drawn, the dynamic obstacle moves into the line of the current trajectory. The algorithm then rewires the trajectory and finds a new collision-free solution, as seen in Figure 5.8c. While the agent moves along the new trajectory, the planning algorithm keeps on expanding the tree, which eventually improves the quality of the trajectory, shown in Figure 5.8d.



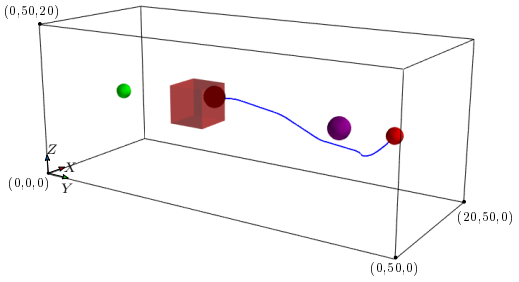
(a) Initial trajectory found by planner.



(b) Rewired trajectory after a few iterations.



(c) First rewiring when obstacle obscured trajectory.



(d) New trajectory after a few iterations of rewiring.

Figure 5.8: The updated trajectory found by the planner at different time steps when one static and one dynamic obstacle are present in the environment.

The experiment shown in Figure 5.8 is then performed 100 times, and the distribution of total distance travelled compared to the initial trajectory found by the planning algorithm is summarised in Table 5.2.

Table 5.2: The comparison between the initial trajectory found by the planning algorithm and the final distance travelled by the agent to reach the goal.

Initial Path		**Final Distance Travelled**	
μ (m)	σ (m)	μ (m)	σ (m)
67.02	15.04	54.11	8.77

From Table 5.2, it is clear that the planning algorithm has the ability to reduce the trajectory length while the agent is moving through the environment, which on average resulted in a 13 m decrease in trajectory length. To determine how much the trajectory cost is reduced, the difference in trajectory length for each experiment was calculated and the distribution thereof is presented in Figure 5.9.

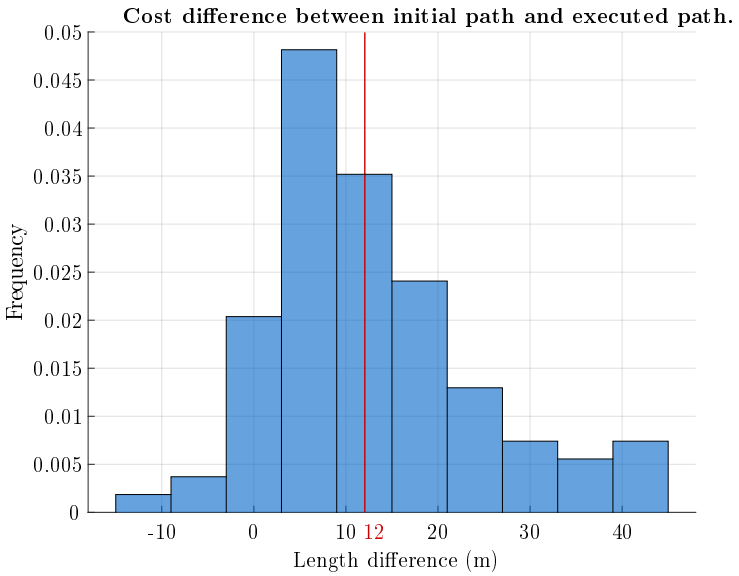


Figure 5.9: The normalised histogram plot of the difference in trajectory length between the initial trajectory found by the planning algorithm and the final distance travelled by the agent.

Figure 5.9 shows that planning algorithm, for most cases, reduced the initial trajectory with an average of 12 m up to a maximum of 44 m, depending on the quality of the initial trajectory. In some cases, the final distance travelled is higher than the initial trajectory, which happened when the cost of the initial trajectory is already low and the planning algorithm has to rewire the trajectory to avoid the dynamic obstacle, increasing the trajectory length. This behaviour is acceptable, seeing as avoiding obstacles have a higher priority than optimizing cost. When looking at all these cases where the total distance travelled is more than the length of the initial trajectory, the total distance travelled never exceeds 50 m, which is below average and therefore can still be considered as a high-quality outcome.

Out of all 100 experiments, the planner was able to avoid the dynamic obstacle 100% of the time and although this is a very simple simulation, it still shows that the planning algorithm was able to successfully rewire the trajectory and find a collision-free solution.

The optimal trajectory length, which is defined as the shortest possible trajectory around the static obstacle, was calculated as 36 m. It was found by taking the minimum-cost trajectory after applying the informed RRT* from Chapter 3, with the manoeuvre-based local planner of Chapter 4, for 10 000 samples (This is not necessarily the true optimal solution, but can be considered as accurate enough). In the ideal case, the total distance travelled by the agent should be as close to the optimal as possible, while still avoiding the dynamic obstacles. The average distance travelled from Table 5.2 is shown to

be 18 m more than the optimal trajectory. The main reason for this deviation is expected to be the avoidance of the dynamic obstacle, but seeing as one of the experiments was able to find a final trajectory of 39 m without collision, the large deviation can not be attributed entirely to obstacle avoidance. From this reason, it was deduced that the cause of the much less optimal trajectories found, was the choice of values chosen for α and β of Equation 3.1 that determines the sampling space of the planner. To increase the convergence rate of the trajectory towards optimality, the planning algorithm should rather focus the sampling in the smaller area of the ellipsoidal subset as well as being forced to sample towards the goal. To determine if this is the case, the experiments are repeated, but β was changed to 100 and α to 0.1. Table 5.3 summarises the results found.

Table 5.3: The distribution of the final distance travelled when α and β was changed to 100 and 0.1 respectively.

Final Distance Travelled	
$\boldsymbol{\mu}$ **(m)**	$\boldsymbol{\sigma}$ **(m)**
49.7	6.88

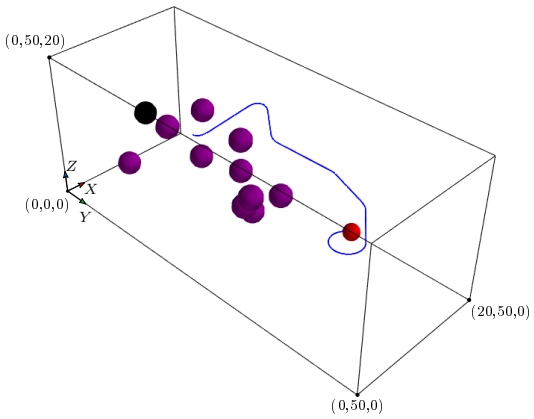
Changing the α and β parameters to decrease the sampling space clearly resulted in higher quality trajectories. The average distance travelled reduced from 54.1 m to 49.7 m and the average difference between the initial trajectory and final distance travelled increased to 18.2 m form the previous 12 m. The drawback of reducing the sampling space is that it limits the exploration of the entire environment, which can result in the planner not being able to avoid dynamic obstacle, because the way around it has not been explored yet. This was the case for 2 out of the 100 experiments for the updated parameter values. It is therefore important to choose these parameter values depending on the needs of the system. In a wide open area with few obstacles, the planner can be much more greedy and sample more in the ellipsoid and less so for a more cluttered environment.

The next simulation will test the motion planning algorithm in a cluttered environment.

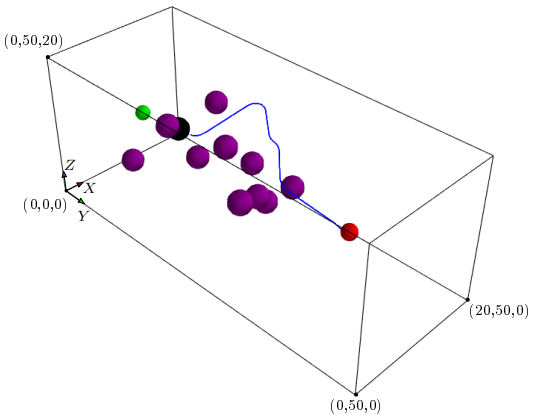
5.3.3.2 Simulation 2: Cluttered Environment

Having shown that the planning algorithm is able to find and update a trajectory in a relatively sparse environment, the next step is to test the planner in a more cluttered environment filled with multiple dynamic obstacles. Simulation 2, shown in Figure 5.10, contains a total of 10 dynamic obstacles obscuring the trajectory from start (10 m, 5 m, 10 m) to goal (10 m, 40 m, 10 m). Each

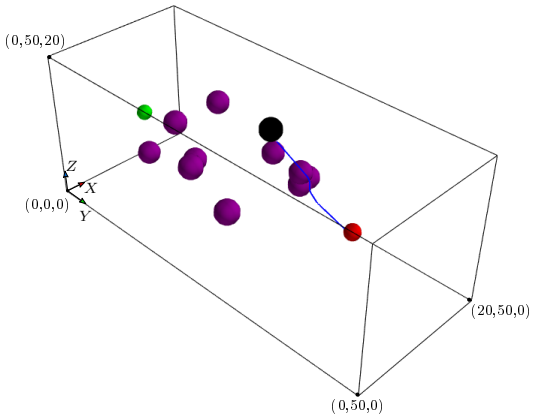
of these obstacles is given a random starting position and a constant velocity in the X-axes (east). The planning algorithm should find a collision-free trajectory through the clutter as well as update the the trajectory to reduce its cost.



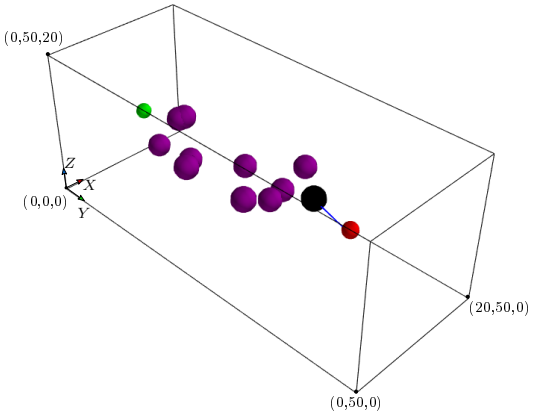
(a) Initial trajectory found by planner.



(b) Rewired trajectory to avoid obstacle.



(c) Collision-free trajectory towards goal.



(d) Final stretch after making it through the clutter.

Figure 5.10: The trajectory found and updated by the planning solution for a cluttered environment with multiple dynamic obstacles.

The purpose of this simulation is to determine if the planner is able to cope with large numbers of obstacles and what influences it had on the performance of planning algorithm. The desired outcome of this simulation would be if the agent was able to move through the clutter without any collisions, while not having travelled a large distance to reach the goal. Such a solution is presented in the four snapshots from Figure 5.10.

The result presented in Figure 5.10 shows the capability of the planning algorithm. The planning algorithm was able to find a safe trajectory through the

clutter with a relatively short distance travelled. This experiment is executed 100 times and the results are shown in Table 5.4.

Table 5.4: The comparison between the initial trajectory and the final distance travelled by the agent for a cluttered environment as well as the average number of collisions that occurred.

Initial Path		**Final Distance Travelled**		**Average Collisions**	
μ (m)	σ (m)	μ (m)	σ (m)	μ	σ
65.51	14.56	47.97	9.78	1.23	0.7

The experiment again shows that the planning algorithm is able to reduce the length of the initial trajectory found. For this experiment, due to no static obstacle, the initial trajectory is reduced with a higher average length of 17.54 m. The major problem that arises with a cluttered environment, is the number of collisions that occurred. The results show that on average, out of a total of 10 obstacles, that a collision occurred with at least one of them, which is not desirable. The total number of collisions that occurred during each experiment is presented in Figure 5.11.

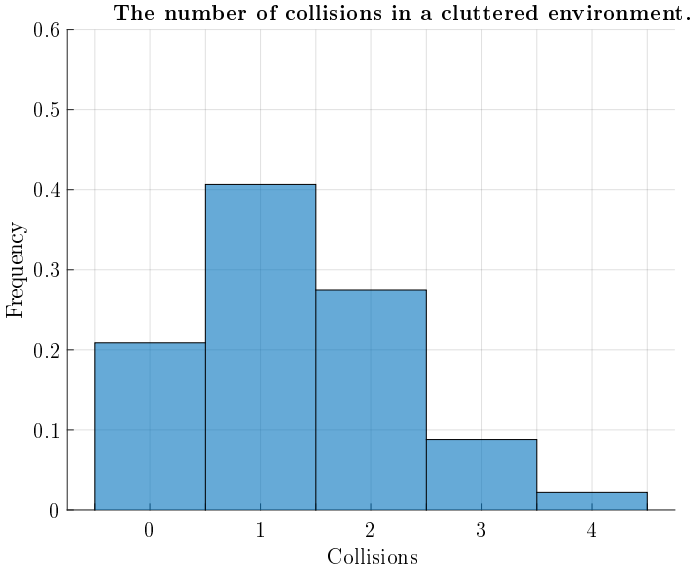


Figure 5.11: The normalised histogram plot of the number of collisions that occurred while trying to move through a cluttered environment.

Figure 5.11 clearly shows that the planning algorithm was only able to avoid all 10 obstacles 20% of time. For 40% of the experiments, there is one collision, 28% for 2 collisions and 12% for more than 2. This environment is unusually cluttered compared to the range of environments in which quad-rotors normally

operate, but it still shows that the planning algorithm is not able to cope with these types of conditions. The reason for the collisions is narrowed down to one of the following issues:

1. The environment has not been explored enough to find collision-free trajectories.
2. The environment has been explored, but the nodes available for rewiring are unreachable, due to the local planner not able to make connections between them.
3. There are too many nodes in the tree, resulting in the planner not able to rewire the right nodes before time runs out.

Issue 1 can be solved by adding more samples in the open regions during the exploration phase, but adding too many nodes may result in issue 3. Issue 2 can also be solved if more nodes are available for rewiring as well as reducing the limitations of the local planner (for example, reducing the turning radius for Dubins trajectories and increasing the constant acceleration for vertical trajectories), but then the quad-rotor might not be able to follow these trajectories with acceptable accuracy. Issue 3 can be seen as the main problem, seeing as both issue 1 and issue 2 can be solved by adding more nodes in the tree. Therefore it was deduced that to improve obstacle avoidance, more computational power is necessary for planning, so that the environment can be completely explored and enough rewiring can be performed during each iteration. It should be noted that the hardware on board a quad-rotor is limited and an alternative method should rather be considered (refer to Appendix C to see what difference more computational power has on the system).

The next simulation tries to solve the issue presented for cluttered environments by executing the planning algorithm, but instead of trying to avoid the obstacles by measuring their current position and re-plan accordingly, while we assume that it is possible to estimate the movement of the obstacles with a Kalman filter [23] and use this information to predict the positions of the obstacles ahead in time. This prediction can then be used by the planner to find collision-free trajectories. This simulation can also be used to determine if the planning algorithm is able to incorporate uncertainty into its calculations, as well as improve upon the current trajectory when new information about the obstacles have been gathered.

5.3.3.3 Simulation 3: Estimation and Prediction

This experiment builds on the previous, by applying the planning algorithm in the same cluttered environment presented in Simulation 2, alongside an

estimator capable of estimating the state space information of the dynamic obstacles in the environment. The estimation was done by assuming that each obstacle has a constant velocity and it is possible to measure its position at any point in time. All obstacles are described by the following state space equation

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{B}\mathbf{u}_k + \mathbf{w}, \tag{5.1}$$

where $\mathbf{x}_{k+1}$ is the state vector with process noise $\mathbf{w}$, and

$$\mathbf{y}_k = \mathbf{C}\mathbf{x}_k + \mathbf{v}, \tag{5.2}$$

where $\mathbf{y}_{k+1}$ is the output vector with measurement noise $\mathbf{v}$. Each obstacle is then presented by the following parameters,

$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ z_k \\ \dot{x_k} \\ \dot{y_k} \\ \dot{z_k} \end{bmatrix} \mathbf{F} = \begin{bmatrix} 1 & 0 & 0 & \tau & 0 & 0 \\ 0 & 1 & 0 & 0 & \tau & 0 \\ 0 & 0 & 1 & 0 & 0 & \tau \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{B} = \mathbf{0}, \quad \mathbf{C} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \tag{5.3}$$

where x_k, y_k and z_k are the position states of the obstacle in the inertial axes with there corresponding velocity states, $\dot{x_k}$, $\dot{y_k}$ and $\dot{z_k}$. The variable τ is the time step of each iteration used by the motion planning algorithm, which is 0.03 s. $\mathbf{w}$ and $\mathbf{v}$ are zero-mean Gaussian distributed white noise is given by,

$$\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}), \tag{5.4}$$

$$\mathbf{v}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R}). \tag{5.5}$$

For the prediction equations of the Kalman Filter, the mean of the estimated obstacle is calculated as

$$\overline{\mathbf{x}}_k = \mathbf{F}\hat{\mathbf{x}}_{k-1} \tag{5.6}$$

and the covariance matrix of the estimated obstacle state is calculated as

$$\overline{\mathbf{P}}_k = \mathbf{F}\hat{\mathbf{P}}_{k-1}\mathbf{F}^T + \mathbf{Q}. \tag{5.7}$$

The measurement update equations of the Kalman filter are given as

$$\mathbf{K}_k = \overline{\mathbf{P}}_k\mathbf{C}^T(\mathbf{C}\overline{\mathbf{P}}_k\mathbf{C}^T + \mathbf{R})^{-1}, \tag{5.8}$$

$$\hat{\mathbf{P}}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{C})\overline{\mathbf{P}}_k, \tag{5.9}$$

and

$$\hat{\mathbf{x}}_k = \overline{\mathbf{x}}_k + \mathbf{K}_k(\mathbf{y}_k - \mathbf{C}\overline{\mathbf{x}}_k). \tag{5.10}$$

By using these equations for each iteration, it is then possible to estimate the states of the dynamic obstacles. Any future position can now be estimated,

with uncertainty, by repeating the prediction equations for the amount of time steps needed to reach a certain point in time.

To make use of the uncertainty about an obstacle after a future state has been predicted, it was assumed that the noise in all three axes are independent from each other. This allows one to calculate the Mahalanobis distance between any node and the mean of the estimated obstacle's position by using the noise covariance matrix $\mathbf{P}$. The Mahalanobis distance [24] is a means to find out how many standard deviations a point in space is away from the mean of a distribution. For a point to be inside the 95% confidence interval of a given distribution, the Mahalanobis distance between the point and the mean should be less than 2.45. This means that if the Mahalanobis distance between the mean of the obstacle, plus the obstacle's size, and the position of a node is less than 2.45, it lies in the 95% confidence region of the obstacle and is seen as a possible collision. Therefore, instead of calculating the Euclidean distance between an obstacle and a node's position to determine if there will be a collision, the 95% confidence region of the obstacle's position is now used. This region changes every time new measurements have been received and the planning algorithm should be able to update its current trajectory accordingly. Refer to Appendix D for the calculations of the Mahalanobis distance.

The planning algorithm, alongside the Kalman Filter estimator, was implemented 100 times for the same environment as shown in Simulation 2 and the position of the dynamic obstacles at each iteration is given to the Kalman filter as measurements. The results received form this simulation are summarized in Table 5.5 below.

Table 5.5: The comparison between the initial trajectory found and the final distance travelled after obstacle estimation was implemented for a cluttered environment when obstacle estimation is added to the system.

Initial Path		**Final Distance Travelled**		**Average Collisions**	
μ **(m)**	σ **(m)**	μ **(m)**	σ **(m)**	μ	σ
59.89	16.17	49.48	10.14	0.0	0.0

This shows that if it is possible to estimate the movement of an obstacle and accurately predict its position after any given time, the planning algorithm is able to find a collision-free trajectory. Out of all 100 simulation, 0 collisions occurred. The planning algorithm is also able to reduce the cost of its initial trajectory with an average of 10.41m, which is less than Simulation 2 that gave an average improvement of 17.54m. This is expected behaviour, because it takes longer to perform the calculations for the Kalman filter estimation, which in turn reduces the number of nodes added to the tree as well as the

amount of rewiring that can be performed. This leads to trajectories that is not as optimal. The uncertainty around obstacles also forces the motion planning solution to take a wider route around the obstacles to compensate for the 95% confidence region. To reduce the travelled distance further, one can again change the α and β parameters, as mentioned previously, and for this case it will not have the same consequences, because of the estimation. With estimation, the trajectory found is seen as collision free, therefore it is not necessary to explore the rest of the environment to be able to avoid obstacles, and thus the focus can be shifted to improve the quality of the current trajectory and not on obstacle avoidance.

The results from Simulation 3 determined that if it is possible to estimate the movement of obstacle in the environment, it will result in much safer trajectories through the environment even with some uncertainty around the position of the obstacle.

The next section will give a summary of the timing information for each major operation, to provide context regarding areas where improvements can be made to increase performance.

5.3.3.4 Timing data

In order to improve the performance of the motion planning solution, it is useful to know which part of the code uses up most of the available computational time. A summary of the operations which is responsible for most of the computational time is presented in Table 5.6 for both Simulation 2 and Simulation 3.

Table 5.6: The per-function breakdown of the time spent in different operations of the motion planning algorithm in both Simulation 2 and Simulation 3.

	Simulation 2	**Simulation 3**
Operation	**% of total time**	**% of total time**
LPM (trajectory cost calculation)	65.81	2.04
Collision detection	26.21	97.24
Neighbours search	0.15	0.03
Sampling	0.12	0.03

The operation that took the most time, is the local planning method, which is tasked to calculate the cost of a trajectory. This is not surprising because every time a node is added, rewired or considered as a parent, the cost to reach the node in question from the starting node has to be calculated from scratch. To reduce the time required by the LPM, these cost calculations need to be optimised or a more efficient LPM method should be designed.

Another operation high on the list, is the collision detection operations for the dynamic and static obstacles. These operations include the collision detection for static obstacles and the colour assignment operations from Section 5.2.1.1. This means that the colour assignment operation took a significant portion of the total processing time. This is a result of the multiple obstacles present in the environment, and again shows why the motion planning solution struggles in Simulation 2 for cluttered environments. The time needed by the collision detection operations drastically increased when estimation was added to the system, which increased from 26.21% of the total time to 97.24%. This shows that estimation methods, like the Kalman filter, are expensive operations, which reduces the quality of the final solution, but as a result are able to reliably find collision free-solutions. To reduce the time required by the collision detection operations, a more effective way to assign a colour to a neighbour should be found.

The rest of the operations present in Table 5.6, takes up only a fraction of the total processing time, which shows that these operations are a lot less expensive.

From the results gathered, it can be concluded that the RT-RRT* as motion planning solution for a quad-rotor, although it has a few shortcomings, is a promising solution for planning in dynamic environment.

Chapter 6

Conclusion and Recommendations

This thesis describes the development and testing of a motion planning algorithm for a rotary-wing UAV to find a collision-free trajectory between any two vehicle states in a predefined environment with dynamic obstacles. The motion planning algorithm implemented made use the Rapidly-exploring random tree (RRT) algorithm, which was altered to make it applicable for three-dimensional, dynamic environments. Alongside the motion planning algorithm, a local planning method (LPM) was integrated, which made it possible for the chosen aircraft to fly the generated trajectories. The viability of the motion planning algorithm was demonstrated using software simulations and the trajectories generated by the planner was determined to be executable by the use of an accurate aircraft model, which was verified during practical tests in research done prior to this thesis.

6.1 Conclusions

This project argued that, of the different types of motion planning algorithms, sampling-based algorithms are the better choice when dynamic environments are considered. The efficient nature of the sampling approach and the ability to remain efficient in higher-dimensional spaces is ideal for more complex problems, which is ideal for the use as a planning algorithm for a rotary-wing UAV.

The most notable contributions of this project, are the implementation of the RT-RRT* as planning algorithm for a quad-rotor, as well as the changes made to improve performance. The existing RT-RRT* was only implemented for two-dimensional spaces with straight line connections as LPM, which is a significantly less complex problem setup. This project shows that this algorithm can be adapted to three-dimensional spaces and that a LPM can be implemented that will generate trajectories that conforms to the dynamic constraints of the quad-rotor. This project also made a significant improvement

to the existing RT-RRT* by changing the rewiring approach, and it was determined that the new colour-based rewiring strategy is a much better approach in terms of the quality of the trajectories found. The use of a simple geometric based manoeuvre set, as LPM, was shown to be an effective way to make the planning algorithm conform to the dynamic constraint of the desired vehicle, and with simple feed-forward control, the vehicle was able to accurately follow any trajectory generated by the planning algorithm.

For a simple, realistic real-world environment, the motion planning solution was able to effectively plan a collision-free trajectory through the environment and avoid all the obstacles. Although the quality of initial trajectories found by the planner was quite low, the planning algorithm was quickly able to reduce the cost and found a much more desirable solution while the agent (quad-rotor) moved through the environment. This is one of the biggest advantages of the RT-RRT* motion planning algorithm compared to others in the same field that usually finds a single trajectory and does not try to change it while the trajectory is being executed, but rather re-plan from scratch.

For a more cluttered environment, the algorithm starts to become less reliable in avoiding dynamic obstacles. Although very cluttered environments are not accurate representations of typical environments quad-rotors will encounter, it clearly shows the limitation of this planning algorithm, which means that to effectively use this planning algorithm in a cluttered environment, much more computational power is needed.

This project also showed that, if it is possible to measure and estimate the movement of obstacles, one can incorporate algorithms like the Kalman Filter, to help with avoiding obstacles more effectively. Making use of obstacle state estimation drastically improved obstacle avoidance, but increased the computational complexity of the algorithm even further. This resulted in much less cost-effective trajectories found. This also means that, if estimation can be performed, the use of the RT-RRT* algorithm might not even be necessary, but a once-of planner could rather be used (for example the informed RRT*), which will yield a much more optimal trajectory if it can be performed before execution of the plan starts. The biggest advantage of using the RT-RRT* algorithm with obstacle estimation with uncertainty, is that the RT-RRT* is able to update and improve the current trajectory when new information about the obstacles are received, which is not the case for a once-off planner seeing as it only relies on the initial information of the environment to plan a trajectory.

The biggest impact on the performance of the planning algorithm is the cost calculation of a trajectory, which is directly related to the complexity of the LPM. This means that, to improve the current performance, one should aim to

improve the effectiveness of the LPM, or choose a completely different LPM. Another major component in the performance of the algorithm, is the collision detection system, which includes the static obstacle collision detection and the colour assignment of nodes. To improve the performance of the planning algorithm, it might be necessary to revisit the collision detection and find a cost-effective solution.

6.2 Recommendations

There are many interesting aspects of the overall motion planning problem that can be investigated that might lead to better performance. The following are some possibilities that can be useful to investigate further and might yield interesting results:

1. The RT-RRT*, and sampling-based motion planning algorithms in general, add many samples to the environment to find and improve a trajectory. Many of these samples are not even used in the final trajectory, or considered during rewiring. The more nodes there are in the tree, the longer it takes to add new nodes and perform rewiring, because the planning algorithm has to look through all the nodes in the tree to find neighbouring node to connect to. This can lead to many different problems that influence the performance of the planner (as mentioned in Chapter 5). Therefore, it might be beneficial to investigate different techniques to effectively remove unwanted nodes to reduce the size of a tree. This might significantly improve the performance of the planner.

2. When considering real-world map representations, obstacles are not perfectly defined and has some uncertainty about its position and size, due to inaccuracies in the sensors used to observe the environment. It will most likely be possible for RT-RRT* and the colour-based rewiring strategy to make use of the uncertainty and plan accordingly when new information about obstacles are received, and update its trajectory to reduce cost. This is a very promising aspect and should be investigated further to determine how well the RT-RRT* will work in these environments.

3. Currently all operations of the motion planning algorithm are done sequentially. This means that the planning algorithm first samples a node, then tries to connect it to other nodes in the tree and only when this operation is completed, it will rewire nodes in the tree to reduce cost. It might be possible to parallelise different operations to increase the overall performance of the planning algorithm and therefore parallelisation should be useful for further investigation.

4. The quad-rotor chosen as test vehicle is rather restrictive, due to its size. It might be useful to use a more manoeuvrable aircraft and test to see if the proposed planning solution can perform better, especially in a cluttered environment.

5. There are much better software products available that is able to accurately simulate a quad-rotor's movement in more realistic environments, which can be a much better way to determine the performance of the planning algorithm. An example of such a software is Gazebo [25]. Gazebo is an open-source simulation environment, which has the ability to accurately simulate robots in complex indoor and outdoor environments. Another possibility to consider is the so-called Robot Operating System (ROS) [26]. ROS is a collection of different libraries and tools which can be used to create complex robot behaviours. ROS has realistic quad-rotor models which can directly be implemented alongside Gazebo for simulation purposes. By implementing the motion planning problem of this project in these software environments, may lead to much more accurate results.

Appendices

Appendix A

SLADe Data and Properties

Table A.1 shows the geometric, inertial and aerodynamic properties of the quad-rotor, SLADe, used as test vehicle for this project.

Table A.1: Numerical values for the symbols that represents the properties associated with the aircraft SLADe [4].

Symbol	Value	Unit	Symbol	Value	Unit
d	0.465	m	I_x	0.5	kg·m^2
r_d	0.18	m	I_y	0.5	kg·m^2
m	15.0	kg	I_z	0.85	kg·m^2
A_D	0.5	m^2	C_D	1.0	
τ	0.125	s			

The feedback gains of the control system described in 2 for the quad-rotor SLADe is presented in Table A.2.

Table A.2: Numerical values of the control feedback gains used for the quad-rotor SLADe [4].

Longitudinal		Lateral		Directional		Heave		Navigation	
K_{q_p}	4.0	K_{P_P}	4.0	K_{r_P}	20.0	K_{w_P}	-20.0	K_N	0.25
K_{q_I}	0.8	K_{P_I}	0.8	K_{r_I}	4.0	K_{w_I}	-2.0	K_N	0.25
K_θ	1.2	K_ϕ	1.2	K_ψ	0.4	K_D	0.3		
K_u	-0.06	K_v	0.06						

Appendix B

Vertical Manoeuvre Intersection Point Calculation

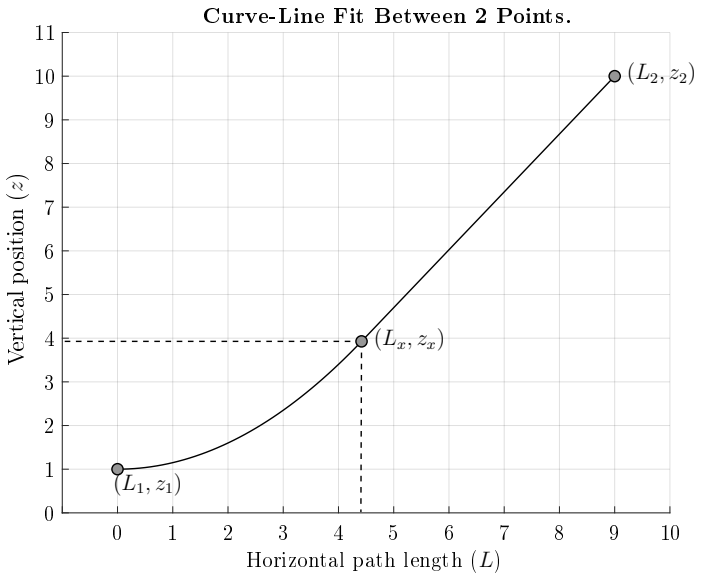


Figure B.1: Fitting a parabola to a line between two given points.

The following assumptions was made to calculate the intersection point between the line and parabola:

- $L1 = 0$, the starting position of the horizontal path length between two states will always be zero.
- The vertical position of both states are given (z_1, z_2).
- The vertical velocity of the starting state is given (V_{z1}).

- The maximum vertical acceleration is constant (A_z).
- The total horizontal path length between the states are given (L_2).
- The horizontal velocity is constant and predefined (V_L).

To calculate the intersection point a parabola is presented as

$$z = aL^2 + bL + c, \tag{B.1}$$

where the first derivative with respect to time (t) is

$$\begin{aligned} \dot{z} &= 2aL\dot{L} + b\dot{L} \\ V_z &= 2aLV_L + bV_L \end{aligned} \tag{B.2}$$

and the second derivative,

$$\begin{aligned} \ddot{z} &= 2a(\dot{L}^2 + L\ddot{L}) + b\ddot{L} \\ A_z &= 2a({V_L}^2). \end{aligned} \tag{B.3}$$

Equation B.3 can then be used to calculate the parabola constant a as

$$a = \frac{A_z}{2{V_L}^2}. \tag{B.4}$$

Substituting B.4 into B.2 gives

$$b = \frac{V_{z1} - 2L_1 a}{V_L}, \tag{B.5}$$

and by substituting B.4 and B.5 into B.1, c can be calculated as

$$c = z1 - a{L_1}^2 - bL_1. \tag{B.6}$$

The line segment from Figure B.1 can be represented as

$$z = eL + f, \tag{B.7}$$

and the first derivative as

$$\begin{aligned} \dot{z} &= e\dot{L} \\ V_z &= eV_L. \end{aligned} \tag{B.8}$$

The line constant e can be calculated as

$$\begin{aligned} eV_L &= 2aL_xV_L + bV_L \\ e &= 2aL_x + b, \end{aligned} \tag{B.9}$$

at intersection point, (L_x, z_x), where the answer to the vertical velocity for the parabola and line are the same.The constant f is then calculated as

$$f = z_2 - eL_2. \tag{B.10}$$

At the intersection point, the parabola and line equations should yield the same value for L, therefore L_x can be solved by

$$a{L_x}^2 + bL_x + c = eL_x + f, \tag{B.11}$$

which gave the answer

$$L_x = L2 \pm \frac{\sqrt{A_z({L_2}^2 A_z + 2{V_L}^2(z_1 - z_2) + 2L_2 V_L V_{z_1})}}{A_z}, \quad 0 \leq L_x \leq L_2. \tag{B.12}$$

z_x can then be calculated by substituting L_x into B.1, which gave

$$z_x = \frac{L_x(2V_L V_{z1} + L_x A_z)}{2V_L^2}. \tag{B.13}$$

To find the intersection point when the velocity at the second state (V_{z_2}) is given, it is needed to calculate the constant acceleration for the parabola section to reach this velocity. The line constant e can be obtained by substituting the velocity at the second state into Equation B.8, which gave

$$e = \frac{V_{z_2}}{V_L}, \tag{B.14}$$

and f is then calculated as

$$f = z_2 - e * L_2. \tag{B.15}$$

The constant acceleration for the parabola section can be calculated by substitution the values of e, f into B.9 and solving the equation, which resulted in

$$\begin{aligned} e &= 2aL_x + b \\ a_z &= \frac{V_L(V_{z1} - V_{z2})^2}{2L_2 V_{z2} + 2V_L z_1 - 2V_L z_2}, \qquad 0 \leq a_z \leq A_z. \end{aligned} \tag{B.16}$$

With the answer to the acceleration, the intersection point can be obtained by substitution a_z into Equation B.2 and solving for L_x, where $V_z = V_2$ due to the line segment having a constant velocity. This gave

$$L_x = \frac{V_L(V_{z2} - V_{z1})}{az}, \qquad L_1 \leq L_x \leq L_2, \tag{B.17}$$

and z_x can then be calculated by substituting L_x and a_z into B.1 as

$$z_x = \frac{V_L z_2 - L_2 V_{z2}}{V_L} + \frac{V_{z2}(V_{z2} - V_{z1})}{az}. \tag{B.18}$$

Appendix C

Computation time increase results

From Simulation 2 in Chapter 5 it was deduced that the main reason why the RT-RRT* struggles in a cluttered environment, is the lack of computational power. To verify if the planning algorithm is able to perform better with more time available for planning, it is decided that for each time step, the planner has 10 times the normal duration to perform the needed operations. This means that for each 0.03 s interval, the planning algorithm is allowed to use 0.3 seconds for expansion and rewiring. This results in an answer that simulates what would happen if the planning algorithm was able to complete the same amount of operations it did for 0.3 seconds in 0.03 seconds. The simulation with more computational time is then executed 100 times, and the results are shown in Table C.1. Form Table C.1 it is clear that, if more computational

Table C.1: The comparison between the initial path found and the final distance travelled by the agent for a cluttered environment with an increase in computation time as well as the average number of collisions that occurred.

Initial Path		**Final Distance Traveled**		**Average Collisions**	
μ (m)	σ (m)	μ (m)	σ (m)	μ (m)	σ (m)
54.73	12.17	46.54	0.49		0.65

power is available, the planning algorithm is able to further decease the cost of the initial path with another 1.43 m. The average number of collisions also decrease form 1.23 to 0.49 collision per simulation and Figure C.1 shows the spread of the amount of collisions that occurred over the 100 simulations.

Figure C.1 shows that the planning algorithm was able to find a safe path 59% of the time. 34% of the time the agent collided with one obstacle and 7% for anything more, up to maximum of 3 collisions.

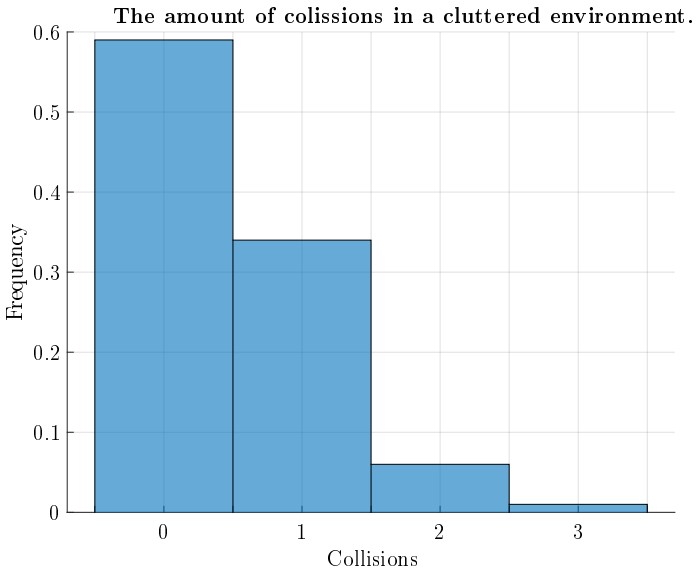


Figure C.1: The normalized histogram plot of the amount of collisions that occurred while trying to move through a cluttered environment when more computation time is given to the planner.

This is a significant better result compared to the one shown in Simulation 2. All though there are still times the planning algorithm was not able to find a collision-free trajectory, it showed that with more computational power, the planning algorithm is able improve obstacle avoidance.

Appendix D

Mahalanobis Distance Calculation

The Mahalanobis Distance is a measurement to find out how many standard deviations a point $\bar{\mathbf{x}}$ is away from the mean of a distribution $\mathbf{S}$ [24]. The Mahalanobis distance for an observation $\bar{\mathbf{x}} = [x_1, x_2, ..., x_N]^T$ from a distribution with a covariance matrix $\mathbf{S}$ and a mean of $\bar{\boldsymbol{\mu}} = [\mu_1, \mu_2, ..., \mu_N]^T$ is defined as

$$d_M = \sqrt{(\bar{\mathbf{x}} - \bar{\boldsymbol{\mu}})^T S^{-1} (\bar{\mathbf{x}} - \bar{\boldsymbol{\mu}})}, \tag{D.1}$$

where, in the case of obstacle estimation, the mean of the obstacle position is represented as $\hat{\mathbf{x}}_k$ and the covariance matrix, $\mathbf{P}$. The observation is the sampled node, $\mathbf{q}_i$ from the motion planning algorithm that is under consideration.

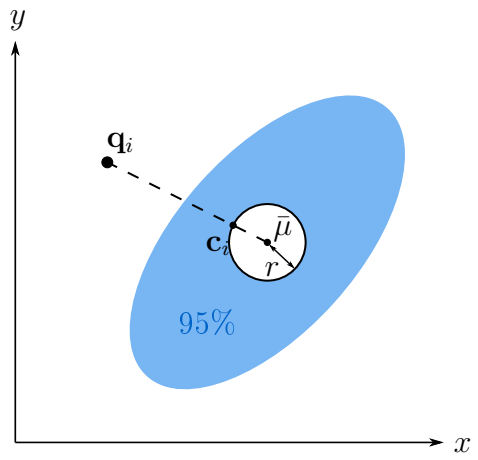


Figure D.1: An example of a typical obstacle with a given size (r), estimated mean ($\boldsymbol{\mu}$) and variance fo a two dimensional environment. The blue ellipse is the 95% confidence region of the obstacle and $\mathbf{q}_i$ is a sampled node.

To calculate if a sampled node, $\mathbf{q}_i$, is inside the 95% confidence region of the distribution around an obstacle, the Mahalanobis distance between the node

and the obstacle should be less that 2.45 for a three-dimensional distribution. Figure D.1 shows a two-dimensional example of a sampled node, $\mathbf{q}_i$, and a circular obstacle with an outer radius of r and an estimated mean of μ. The blue elliptical region represents the 95% confidence region of the obstacles position. The point $\mathbf{c}_i$, which lies on the edge of the obstacle can be calculated as

$$\mathbf{c}_i = \bar{\mu} + \frac{r(\bar{q} - \bar{\mu})}{\|\bar{q} - \bar{\mu}\|}. \tag{D.2}$$

Point $\mathbf{c}_i$, can now be used as the new mean in Equation D.1 and $\mathbf{q}_i$ the observation point. If the answer yields a value smaller than 2.45, then the point $\mathbf{q}_i$ lies inside the 95% confidence region of the obstacle, and will be seen as a collision.

List of References

[1] van Daalen, C.E.: *Conflict Detection and Resolution for Autonomous Vehicles.* Ph.D. thesis, Stellenbosch University, 2010.

[2] Engelbrecht, J.A.A.: Advance automation 833, introductory course to aircraft dynamics. April 2018.

[3] Elbanhawi, M.: Sampling-based robot motion planning: A review. *IEEE Access*, vol. 2, pp. 56–77, Jan 2014.

[4] Peddle, I.K.: Slade (quad) project control and simulation report. 2009.

[5] Amazon primeair. `https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011`. Accessed: 2019-10-28.

[6] Peddle, I.K.: SLADe - technical development report. 31 July 2007.

[7] LaValle, S.M.: *Planning Algorithms.* Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[8] LaValle, S.M.: *Combinatorial Motion Planning*, pp. 250–310. Cambridge University Press, 2006.

[9] Uyanik, K.F.: A study on artificial potential fields. 2011.

[10] LaValle, S.M.: *Sampling-Based Motion Planning*, pp. 185–248. Cambridge University Press, 2006.

[11] Kavraki, L.E., Svestka, P., Latombe, J.. and Overmars, M.H.: *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*, vol. 12. Aug 1996.

[12] Hsu, D., Latombe, J.-C. and Kurniawati, H.: On the probabilistic foundations of probabilistic roadmap planning. *IJJR*, vol. 25, pp. 627–643, July 2006.

[13] LaValle, S.M.: *Rapidly-exploring random trees: A new tool for path planning.* Iowa State University, 1998.

[14] LaValle, S.M. and Kuffner, J.J.: Randomized kinodynamic planning. *IJJR*, vol. 20, pp. 378–400, May 2001.

[15] AMATO, N.M. and Wu, Y.: A randomized roadmap method for path and manipulation planning. *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, pp. 113–120, April 1996.

[16] Karaman, S. and Frazzoli, E.: Incremental sampling-based algorithms for optimal motion planning. 2010.

[17] Gammell, J.D., Srinivasa, S.S. and Barfoot, T.D.: Informed RRT*: Optimal incremental path planning focused through an admissible ellipsoidal heuristic. 2014.

[18] Naderi, K., Rajamäki, J. and Hämäläinen, P.: RT-RRT*: A real-time path planning algorithm based on RRT*. In: *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, MIG '15, pp. 113–118. ACM, 2015.

[19] Webb, D.J. and van den Berg, J.: Kinodynamic RRT*: Optimal motion planning for systems with linear differential constraints. 2012.

[20] Dubins, L.E.: On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, vol. 79, pp. 497–516, 1957.

[21] Cython C-extensions for python. `https://cython.org/`. Accessed: 2018-05-12.

[22] Mayavi: 3D scientific data visualization and plotting in python. `https://docs.enthought.com/mayavi/mayavi/`. Accessed: 2019-03-09.

[23] Steyn, W.H., Peddle, I.K. and van Daalen, C.E.: Control systems 414, the kalman filter(state estimation). Feb 2017.

[24] McLachlan, G.J.: Mahalanobis distance. *Department of Mathematics, The University of Oueensland.*, 1999.

[25] Gazebo: Robot simulation made easy. `http://gazebosim.org/`. Accessed: 2019-10-28.

[26] Robot operating system (ROS). `https://www.ros.org/about-ros/`. Accessed: 2019-10-28.